



## Calhoun: The NPS Institutional Archive DSpace Repository

---

Theses and Dissertations

1. Thesis and Dissertation Collection, all items

---

2010-12

# IP services design and implementation in a prototype device for transient tactical access to sensitive information

Yoong, Ho Liang.

Monterey, California. Naval Postgraduate School

---

<http://hdl.handle.net/10945/4982>

---

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

*Downloaded from NPS Archive: Calhoun*



<http://www.nps.edu/library>

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community.

Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School  
411 Dyer Road / 1 University Circle  
Monterey, California USA 93943



**NAVAL  
POSTGRADUATE  
SCHOOL**

**MONTEREY, CALIFORNIA**

**THESIS**

**IP SERVICES DESIGN AND IMPLEMENTATION IN A  
PROTOTYPE DEVICE FOR TRANSIENT TACTICAL  
ACCESS TO SENSITIVE INFORMATION**

by

Ho Liang Yoong

December 2010

Thesis Advisor:  
Second Reader:

Cynthia E. Irvine  
David J. Shifflett

**Approved for public release; distribution is unlimited**

**THIS PAGE INTENTIONALLY LEFT BLANK**

**REPORT DOCUMENTATION PAGE**
*Form Approved OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

<b>1. AGENCY USE ONLY (Leave blank)</b>	<b>2. REPORT DATE</b> December 2010	<b>3. REPORT TYPE AND DATES COVERED</b> Master's Thesis
<b>4. TITLE AND SUBTITLE</b> IP Services Design and Implementation in a Prototype Device for Transient Tactical Access to Sensitive Information		<b>5. FUNDING NUMBERS</b>
<b>6. AUTHOR(S)</b> Ho Liang Yoong		<b>8. PERFORMING ORGANIZATION REPORT NUMBER</b>
<b>7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)</b> Naval Postgraduate School Monterey, CA 93943-5000		<b>10. SPONSORING/MONITORING AGENCY REPORT NUMBER</b>
<b>9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)</b> N/A		
<b>11. SUPPLEMENTARY NOTES</b> The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number _____ N.A._____.		
<b>12a. DISTRIBUTION / AVAILABILITY STATEMENT</b> Approved for public release; distribution is unlimited		<b>12b. DISTRIBUTION CODE</b>

**13. ABSTRACT (maximum 200 words)**

In network-centric warfare, access to critical information can result in a strategic advantage. During critical situations, a soldier using tactical devices may need transient access to information beyond their normal clearances. The Least Privilege Separation Kernel (LPSK) being developed at the Naval Postgraduate School, can be the basis of an extended multilevel security (MLS) system that can support and control such access. A Trusted Services Layer (TSL), which depends on the LPSK, provides support for various multilevel security services. Currently, the LPSK lacks a software network stack for networking communications. Without networking functionality, tactical devices cannot share vital situational updates and information superiority is unattainable.

An Internet Protocol (IP) stack was proposed for the LPSK-based system. The IP stack is to be implemented in the context of the LPSK architecture, which uses modularity and layering to organize its software. Open source implementations of the IP stack were evaluated to leverage the common functionality required by all IP stacks. Lightweight Internet Protocol (LWIP) was selected as a starting point for use with the LPSK. LWIP required modifications for use with the LPSK. The IP stack and a proof of concept networking demonstration were successfully implemented in this project.

<b>14. SUBJECT TERMS</b> least privilege separation kernel, multilevel security, lightweight internet protocol, network protocol stack, internet protocol stack		<b>15. NUMBER OF PAGES</b> 139
		<b>16. PRICE CODE</b>
<b>17. SECURITY CLASSIFICATION OF REPORT</b> Unclassified	<b>18. SECURITY CLASSIFICATION OF THIS PAGE</b> Unclassified	<b>19. SECURITY CLASSIFICATION OF ABSTRACT</b> Unclassified
		<b>20. LIMITATION OF ABSTRACT</b> UU

**THIS PAGE INTENTIONALLY LEFT BLANK**

**Approved for public release; distribution is unlimited**

**IP SERVICES DESIGN AND IMPLEMENTATION IN A PROTOTYPE DEVICE  
FOR TRANSIENT TACTICAL ACCESS TO SENSITIVE INFORMATION**

Ho Liang Yoong

Civilian, Singapore Technologies Engineering, Singapore

B.Eng., Nanyang Technological University, Singapore, 2001

M.Sc., Singapore MIT Alliance, National University of Singapore, Singapore, 2002

Submitted in partial fulfillment of the  
requirements for the degree of

**MASTER OF SCIENCE IN COMPUTER SCIENCE**

from the

**NAVAL POSTGRADUATE SCHOOL  
December 2010**

Author: Ho Liang Yoong

Approved by: Cynthia E. Irvine  
Thesis Advisor

David J. Shifflett  
Second Reader

Peter J. Denning  
Chairman, Department of Computer Science

**THIS PAGE INTENTIONALLY LEFT BLANK**

## **ABSTRACT**

In network-centric warfare, access to critical information can result in a strategic advantage. During critical situations, a soldier using tactical devices may need transient access to information beyond their normal clearances. The Least Privilege Separation Kernel (LPSK), being developed at the Naval Postgraduate School, can be the basis of an extended multilevel security (MLS) system that can support and control such access. A Trusted Services Layer (TSL), which depends on the LPSK, provides support for various multilevel security services. Currently, the LPSK lacks a software network stack for networking communications. Without networking functionality, tactical devices cannot share vital situational updates and information superiority is unattainable.

An Internet Protocol (IP) stack was proposed for the LPSK-based system. The IP stack is to be implemented in the context of the LPSK architecture, which uses modularity and layering to organize its software. Open source implementations of the IP stack were evaluated to leverage the common functionality required by all IP stacks. Lightweight Internet Protocol (LWIP) was selected as a starting point for use with the LPSK. LWIP required modifications for use with the LPSK. The IP stack and a proof of concept networking demonstration were successfully implemented in this project.

**THIS PAGE INTENTIONALLY LEFT BLANK**

## TABLE OF CONTENTS

I.	INTRODUCTION.....	1
A.	MOTIVATION .....	2
B.	OBJECTIVES .....	3
C.	THESIS ORGANIZATION.....	3
II.	BACKGROUND .....	5
A.	NETWORK TCP/IP PROTOCOLS .....	5
1.	TCP/IP Protocol Suite .....	5
2.	Encapsulation and Demultiplexing .....	8
3.	Data Packet Formats .....	10
a.	<i>TCP Segment</i> .....	11
b.	<i>UDP Datagram</i> .....	13
c.	<i>IP Datagram</i> .....	13
d.	<i>ICMP Message</i> .....	15
e.	<i>Ethernet Frame</i> .....	16
B.	LEAST PRIVILEGE SEPARATION KERNEL.....	18
1.	Separation Kernels.....	18
2.	Least Privilege Separation Kernel.....	20
3.	Trusted Computing Exemplar Project .....	21
III.	REQUIREMENTS AND DESIGN.....	23
A.	REQUIREMENTS.....	23
1.	Functional Requirements for the System Prototype.....	23
2.	Functional Requirements for the IP Protocol Stack.....	24
B.	DESIGN CONSIDERATIONS.....	27
1.	High Level IP Protocol Stack Design .....	27
2.	IP Protocol Stack in the TCX LPSK.....	28
3.	Layering in the TCX LPSK .....	29
4.	Open Source Network Protocol Stacks .....	30
5.	Lightweight Internet Protocol (LWIP) .....	32
6.	LWIP Data Structures.....	33
a.	<i>Packet Buffer (pbuf)</i> .....	33
b.	<i>Network Interface (netif)</i> .....	34
c.	<i>ICMP Echo Header (icmp_echo_hdr)</i> .....	35
d.	<i>IP Header (ip_hdr)</i> .....	35
e.	<i>Ethernet Header (eth_hdr)</i> .....	36
f.	<i>ARP Header (etharp_hdr)</i> .....	37
7.	LWIP Functions and Function Call Flow.....	37
8.	Final Design .....	43
C.	SUMMARY .....	48
IV.	IMPLEMENTATION .....	49
A.	DEVELOPMENT ENVIRONMENT .....	49
B.	IMPLEMENTATION METHODOLOGY .....	49

1.	Configuring the LPSK for LWIP .....	49
2.	Miscellaneous System Functions .....	50
3.	Modification of LWIP Functions.....	51
4.	Implementation of the Ping Application.....	52
C.	SUMMARY .....	55
V.	TESTING.....	57
A.	FUNCTIONAL TEST CASES .....	57
1.	Original LWIP Functions .....	57
2.	Newly Added and Modified LWIP Functions .....	58
B.	ACCEPTANCE TESTS .....	77
1.	Acceptance Test A1 and Results.....	77
2.	Acceptance Test A2 and Results.....	77
3.	Acceptance Test A3 and Results.....	78
C.	PROBLEMS ENCOUNTERED .....	79
1.	Memory Allocation .....	79
2.	LWIP Heap Memory .....	80
D.	SUMMARY .....	80
VI.	CONCLUSION AND FUTURE WORK .....	81
A.	CONCLUSION .....	81
B.	FUTURE WORK .....	82
1.	Separation of Privilege Levels.....	82
2.	Transport Layer Services Support.....	82
3.	Network Layer Services Extension.....	83
APPENDIX A:	MAKEFILE CONFIGURATION.....	85
APPENDIX B:	TEST PROCEDURES.....	87
LIST OF REFERENCES.....		115
INITIAL DISTRIBUTION LIST .....		119

## LIST OF FIGURES

Figure 1.	TCP/IP Protocol Suite layers and sample protocols (From [5]) .....	6
Figure 2.	TCP/IP Protocol Suite protocols used in this work (After Figure 1.4 in [3]) ....	7
Figure 3.	Data encapsulation in the protocol stack (From Figure 1.7 in [3]) .....	9
Figure 4.	Demultiplexing of received ethernet frame (After Figure 1.8 in [3]) .....	10
Figure 5.	TCP segment in IP datagram (From Figure 17.1 in [3]).....	12
Figure 6.	TCP header (From Figure 17.2 in [3]) .....	12
Figure 7.	UDP datagram in IP datagram (From Figure 11.1 in [3]).....	13
Figure 8.	UDP header (From Figure 11.2 in [3]) .....	13
Figure 9.	IP header (From Figure 3.1 in [3]).....	14
Figure 10.	ICMP messages in IP datagram (From Figure 6.1 in [3]).....	15
Figure 11.	ICMP message (From Figure 6.2 in [3]).....	15
Figure 12.	ICMP type message format for Echo Request and Reply (From [11]).....	16
Figure 13.	Ethernet encapsulation (From Figure 2.1 in [3]) .....	17
Figure 14.	ARP Request/Reply packet format (From [11]) .....	17
Figure 15.	LPSK configuration (From Figure 1 in [16]).....	20
Figure 16.	High level IP protocol stack design .....	28
Figure 17.	Architecture for the TCX LPSK (After Figure 1 in [21]) .....	29
Figure 18.	Modules and layering in the TCX LPSK .....	29
Figure 19.	LWIP function call flow for sending and receiving IP packets .....	41
Figure 20.	Flow sequence for sending and receiving IP packets .....	42
Figure 21.	IP protocol stack final design.....	45
Figure 22.	Network topology for Net_Prod and Net_Cons partitions .....	53

**THIS PAGE INTENTIONALLY LEFT BLANK**

## LIST OF TABLES

Table 1.	Inter-partition flow policy (From Table 1 in [16]).....	19
Table 2.	Subject-Resource flow matrix (From Table 3 in [16]) .....	21
Table 3.	Functional requirements for the system prototype.....	24
Table 4.	Functional requirements for the IP protocol stack.....	25
Table 5.	Functional requirements for network layer services (FR1-1) .....	25
Table 6.	Functional requirements for ethernet layer services (FR1-2) .....	26
Table 7.	Functional requirements for ping application (FR2-1) .....	26
Table 8.	Evaluation criteria of network protocol stacks .....	31
Table 9.	TCP/IP features implemented by uIP and LWIP (From Table 1 in [27]).....	32
Table 10.	Packet buffer data structure (From [27]) .....	33
Table 11.	Network interface data structure (From [29]).....	34
Table 12.	ICMP echo header data structure (From source code in [30]).....	35
Table 13.	IP header data structure (From source code in [30]).....	36
Table 14.	Ethernet header data structure (From source code in [30]).....	36
Table 15.	ARP header data structure (From source code in [30]) .....	37
Table 16.	LWIP functions (From [24]) .....	39
Table 17.	Flow sequence for sending and receiving IP packets .....	43
Table 18.	Data structure for IP datagram buffer .....	46
Table 19.	Data structure for Ethernet frame buffer.....	47
Table 20.	Essential LWIP files used .....	50
Table 21.	Miscellaneous system functions required by LWIP .....	51
Table 22.	Input parameters of additional functions .....	54
Table 23.	Return values between functions .....	55
Table 24.	Testing original LWIP functions .....	58
Table 25.	Function test groupings.....	59
Table 26.	Function test Group A – low_level_output() .....	60
Table 27.	Function test Group B – get_next_frame() .....	62
Table 28.	Function test Group C – get_eth_ip_frame() .....	63
Table 29.	Function test Group D – get_ip_datagram() .....	65
Table 30.	Function test Group E – get_icmp_packet() .....	67
Table 31.	Function test Group F – recv_ping().....	68
Table 32.	Function test Group G – get_eth_arp_frame() .....	69
Table 33.	Function test Group H – ip_input_verify() .....	71
Table 34.	Function test Group I – etharp_request() .....	72
Table 35.	Function test Group J – send_icmp() .....	74
Table 36.	Function test Group K – send_ping().....	76
Table 37.	Acceptance test A1: Sending ICMP Echo from one host .....	77
Table 38.	Acceptance test A2: Sending ICMP Echo between hosts.....	78
Table 39.	Acceptance test A3: Sending ICMP Echo to non-responding host .....	79

**THIS PAGE INTENTIONALLY LEFT BLANK**

## **LIST OF ACRONYMS AND ABBREVIATIONS**

ARP	Address Resolution Protocol
DNS	Domain Name System
FR	Functional Requirements
FTP	File Transfer Protocol
HTTP	HyperText Transfer Protocol
ICMP	Internet Control Message Protocol
IGMP	Internet Group Management Protocol
IP	Internet Protocol
LPSK	Least Privilege Separation Kernel
LWIP	Lightweight Internet Protocol
OS	Operating System
OSPF	Open Shortest Path First
POP3	Post Office Protocol version 3
RARP	Reverse Address Resolution Protocol
RFC	Request for Comments
SMTP	Simple Mail Transfer Protocol
SR	System Requirements
SSH	Secure Shell
TCP	Transmission Control Protocol
TCX	Trusted Computing Exemplar
TPA	Trusted Path Application
TSL	Trusted Services Layer
UDP	User Datagram Protocol
uIP	Micro Internet Protocol

**THIS PAGE INTENTIONALLY LEFT BLANK**

## **ACKNOWLEDGMENTS**

I would like to thank my advisor, Dr. Cynthia Irvine, for her valuable guidance and support for the completion of this thesis. I would also like to thank David Shifflett, for his technical expertise and insights as a second reader. I would like to express my gratitude to Singapore Technologies Engineering for sponsoring me on this master course. Last, but not least, I thank my wife, Calyn, for her patience and understanding throughout the thesis process.

**THIS PAGE INTENTIONALLY LEFT BLANK**

## I. INTRODUCTION

In today's network-centric warfare, having an information advantage is critical to any successful mission. Tactical wearable devices permit soldiers on the ground to receive and share information, allowing them situational awareness. There are, however, concerns regarding access to and protection of information when it is at different classification levels. These concerns must be considered during critical situations when soldiers may need to access information at a higher classification level than they are normally cleared for. Information labeled Top Secret may be disseminated to a tactical device that is assigned to soldier who is cleared only to the Secret level. The Top Secret information, if not properly handled and securely protected by the tactical device, would be accessible by the soldier even during non-critical times. Moreover, if the tactical device is unable to protect classified information and falls into the hands of adversary able to access it, the battle may be lost. Alternatively, if the tactical device does not grant soldiers access to sensitive information at critical moments, soldiers may be less informed of any situational updates, which may be crucial to winning the battle. Therefore, a tactical multilevel security (MLS) device that is high assurance is required to protect information at different classification levels and to facilitate transient access to critical information at designated times. A Least Privilege Separation Kernel (LPSK) is able to meet such requirement since it supports the notion of transient trust whereby sensitive information is temporarily available to users only during critical situations [1].

The Trusted Computing Exemplar (TCX) project spearheaded by the Center for Information Systems Security Studies and Research (CISR) at the Naval Postgraduate School (NPS) provides an “example of how high assurance trusted computing components can be built” [2]. The LPSK is part of the TCX project. It will be used as a component for a tactical MLS system prototype. LPSK provides the assurance foundation for a tactical MLS system prototype. It uses the principle of *least privilege* and the concept of subject-resource flows, which is an extension of inter-partition flow concept of *separation kernels*. The LPSK project also uses extensive modularity and layering in its implementation.

The MLS services are supported in a Trusted Services Layer (TSL), which depends on the LPSK. The MLS system prototype, however, lacks an Internet Protocol (IP) stack for interacting with network devices. A tactical MLS device with networking capabilities is essential in the battlefield. Soldiers in the battlefield who carry a tactical MLS device without networking capabilities are likely to be on the losing side. Those soldiers would not be able to receive updated situational information and would be forced to fight the battle at an information disadvantage. Having an IP stack in the MLS system prototype not only allows communication with network devices, but also opens the possibility of establishing connections to the Internet. The motivation for considering an IP stack implementation for the tactical MLS system is elaborated in the next section.

## A. MOTIVATION

The current services provided in conjunction with the MLS system prototype do not include a software-based network protocol stack. Neither does the system prototype have a hardware network driver. Writing network drivers for hardware is difficult. Therefore, as an interim solution prior to the development of a complete hardware driver module, building an IP stack supported by functions that simulate network traffic is the motivation for this thesis project.

Having an IP stack in the MLS system prototype enables several interim and long term capabilities. Network traffic in the form of IP data packets can be simulated using the IP stack. A software module can be implemented in the prototype to simulate the generation of IP packets of a physical hardware device. Using the generated IP packets, experiments can be conducted to study the implications of various types of IP packets coming into the prototype. In addition, considerable functionality can be explored using simulated network traffic.

The need for an IP stack leads to the postulation that the IP stack can be built and used with supporting software to simulate network traffic in a tactical MLS system prototype. It is also hypothesized that the design of an IP stack can be modularized and layered in a manner similar to the overall LPSK architecture, thus contributing to arguments that it is a high assurance system.

## **B. OBJECTIVES**

The main focus of this study is to implement an IP stack in the MLS system prototype. A further objective of this thesis is to provide a producer-consumer demonstration where both the producer and the consumer depend upon the IP stack and networked communication to interoperate.

## **C. THESIS ORGANIZATION**

Chapter I provides an introduction to this thesis and includes the motivation and objectives of this work. Chapter II provides background information on the key concepts of TCP/IP protocols and the LPSK. Chapter III describes the requirements and design considerations for the IP protocol stack. Chapter IV explains implementation details for the IP protocol stack, including a description of the software and hardware environment as well as the methodology used in the development of the IP protocol stack. Chapter V describes the functional and acceptance test cases for testing the IP protocol stack, and a discussion of problems encountered. Chapter VI concludes the thesis with recommendations for future enhancement of the IP protocol stack.

**THIS PAGE INTENTIONALLY LEFT BLANK**

## **II. BACKGROUND**

Before discussing the implementation of an IP stack for the LPSK, it is useful to review how the networking stack is organized and the system for which it will be implemented. Section A examines the key concepts of the TCP/IP protocols. Section B elaborates on the Least Privilege Separation Kernel (LPSK) implemented in the TCX project [2].

### **A. NETWORK TCP/IP PROTOCOLS**

This section provides a brief review of the TCP/IP stack. Readers interested in more details should consult Steven's TCP/IP Illustrated [3].

#### **1. TCP/IP Protocol Suite**

The set of communication protocols commonly used for communicating with other systems in a network is the Internet Protocol Suite. It is also called the TCP/IP Protocol Suite where Transmission Control Protocol (TCP) and Internet Protocol (IP) are the two most important protocols. TCP ensures that data is reliably exchanged between two hosts. IP manages the addressing and routing facet of communication within a network and across networks.

Usually, communication protocol suites are made up of a set of software layers. Each layer involves a set of functions and operations. A lower layer exports a set of resources and functionalities to an upper layer, where they are used to create new abstractions and functionalities. For example, the lower layer translates data packets passed down from an upper layer into frames, which eventually will be transmitted through physical devices to the network. In contrast, the upper layer handles data passed up from the lower layer and is closer to the user or application interface. The TCP/IP Protocol Suite has four layers in accordance with RFC1122 [4]. The layers from lowest to highest are: Network Interface Layer, Network Layer, Transport Layer and Application Layer. Examples of common protocols in each layer are shown in Figure 1.

<b>Application Layer</b>	DNS, FTP, HTTP, POP3, SMTP, SSH, Telnet, Echo
<b>Transport Layer</b>	TCP, UDP
<b>Network Layer</b>	IP, ICMP, IGMP
<b>Network Interface Layer</b>	ARP, RARP, OSPF

Figure 1. TCP/IP Protocol Suite layers and sample protocols (From [5])

Each layer has its own set of responsibilities in the overall communications protocol. The network interface layer, also known as data-link layer, includes the management of the device driver and the network interface card within the host. This layer also provides an interface to the hardware that is physically attached to a host. The network layer is responsible for managing the transport of packets from a source host across the network to a specific destination host, based on the destination network address. The transport layer is responsible for the flow of data between two hosts in support of the application layer. The application layer contains the functionality for each particular application.

The TCP/IP protocol suite consists of various protocols. Figure 2 shows the protocols used in this study.

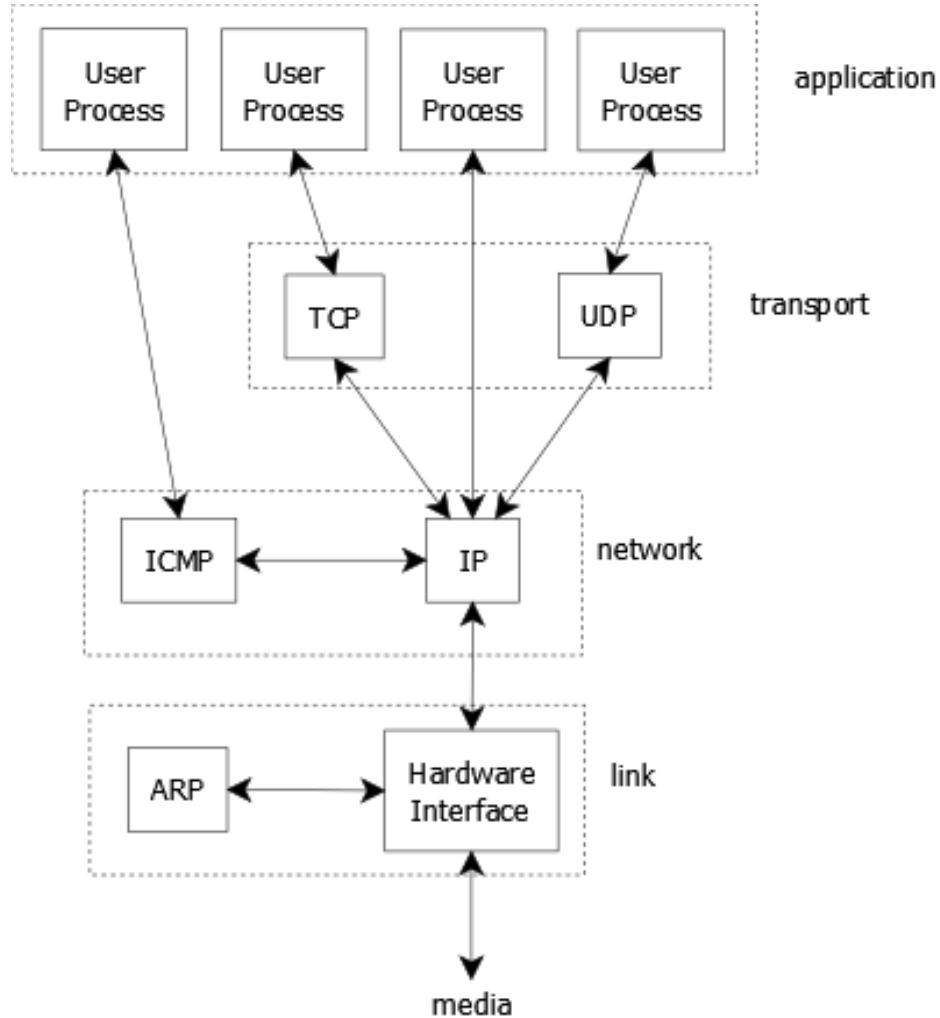


Figure 2. TCP/IP Protocol Suite protocols used in this work (After Figure 1.4 in [3])

In the transport layer, TCP and the User Datagram Protocol (UDP) are the two most common transport protocols. TCP is a connection-oriented protocol and ensures reliable delivery of data between two hosts. *Connection-oriented* [6] means that a stream of data is received in the same order as it was sent. TCP has flow control mechanisms designed to provide guaranteed delivery. As such, there is a performance overhead involved for TCP due to its flow control mechanism. On the other hand, UDP does not guarantee that the data it is delivering will reach its final destination. UDP has a smaller overhead and is used where speed is required and reliability is not needed, for example, in streaming media.

In the network layer, Internet Protocol (IP) is the main protocol used. Any data from TCP and UDP or data received from the network interface layer will go through IP. Internet Control Message Protocol (ICMP) is in the same layer as IP. ICMP is mainly used by IP for exchanging error messages with the network layer of another host. ICMP can also be accessed by applications such as *ping* for diagnostic purposes.

The Address Resolution Protocol (ARP) is a protocol at the network interface layer. ARP is used to map the IP address in the network layer to the hardware address (MAC address) used by network interface. The MAC address of a network interface card is assigned by the manufacturer and is stored in the hardware. Since the ARP provides a mapping of a MAC address to an IP address, the MAC address of a network interface card can be determined by using an ARP query with the IP address assigned to the system. When a local host broadcasts an ARP request, the remote host that belongs to the IP address requested will send back an ARP reply. Both the local host and the remote host will add an entry to their ARP cache table to store the IP-MAC address mapping. The ARP cache entry can be either static or dynamic. A static entry is manually added and stays permanent. A dynamic entry is added automatically and stays valid for a specified period.

When data is sent over an internetwork from one host to another, it may originate in a high layer of the TCP/IP stack and may have to travel across more than one physical network. In order for the data to be delivered and received correctly, it is encapsulated with more information as it flows down the TCP/IP protocol stack. On the receiving end, the received bit stream will be demultiplexed as it flows up the protocol stack. The next section will elaborate on the encapsulation and demultiplexing process of the TCP/IP protocol.

## **2. Encapsulation and Demultiplexing**

When a user application sends data over the network, the data is sent through different layers in the TCP/IP protocol stack. Each unit of data sent from one layer to the next layer is of a different type. The data unit that the transport layer sends to the network layer is called a *segment*; and network layer to network interface layer is called a

*datagram*. The bit stream flowing through an Ethernet is called a *frame*. An Ethernet is a type of bus network based on the IEEE 802.3 standard [7]. It is widely used for local area networks (LAN). It works on an access control method called Carrier Sense, Multiple Access with Collision Detection (CSMA/CD) [8]. The access method allows multiple computers in a network to send data at the same time, and has a mechanism for detecting collisions that happen when two computers send data to one another at the same instant.

Figure 3 shows the data encapsulation process. As data passes through each layer, header information of the corresponding layer is prepended to it. For instance, when a user sends data to a destination host from a user application, the user data is prepended with an application header as it passes through the application layer. From the transport layer to the network layer, TCP or UDP header information is added to the application data. IP header information, which is added to the TCP or UDP segment, is passed down to the network interface layer as an IP datagram. An Ethernet header is then prepended to the IP datagram as data travels through network interface layer. Eventually, the data is physically transmitted through the Ethernet link as a stream of bits.

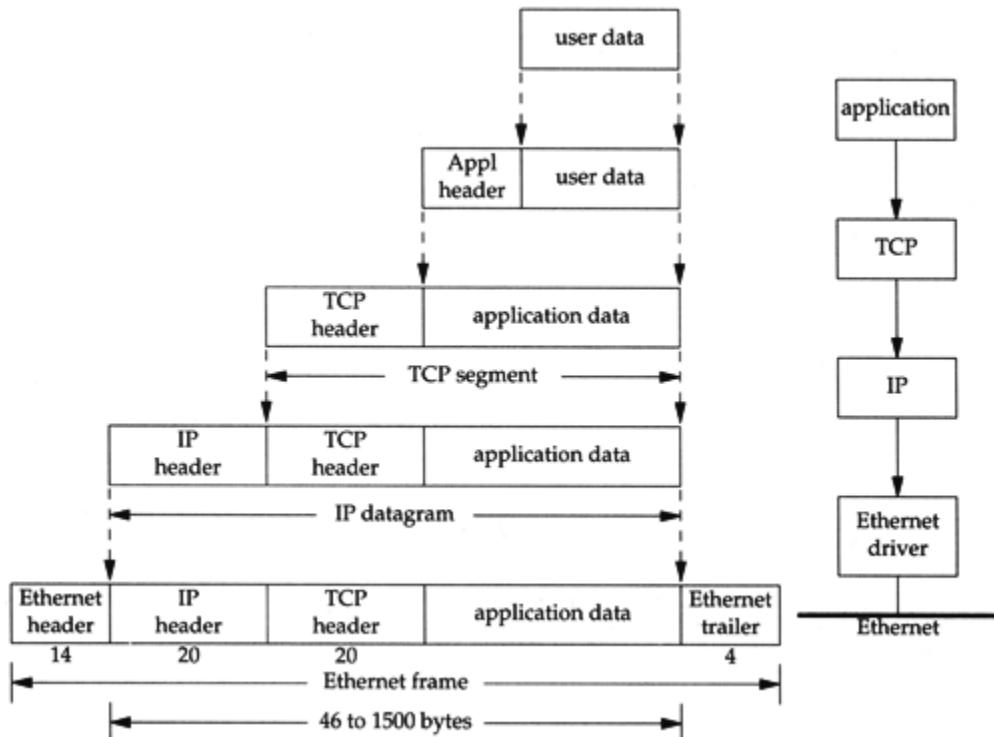


Figure 3. Data encapsulation in the protocol stack (From Figure 1.7 in [3])

Upon receiving the bit stream on the destination host, the encapsulation process is reversed. This process is known as *demultiplexing*. Figure 4 shows the demultiplexing process when an Ethernet frame is received. The network interface layer removes the Ethernet header from the Ethernet frame and passes it up to the network layer as an IP datagram. The IP layer removes the IP header from the datagram and passes the TCP or UDP segment to transport layer. The transport layer then removes the TCP or UDP header and passes it up the protocol stack for use by the user application. In Figure 4, the ICMP protocol is drawn slightly above IP protocol and below TCP/UDP. This is to illustrate that ICMP does not belong to the transport layer but rather, it is an addition to IP. This means that ICMP messages are encapsulated in IP datagrams and both of them belong to the network layer. Similarly for the ARP protocol, ARP has its own Ethernet frame type, so it is below the IP layer.

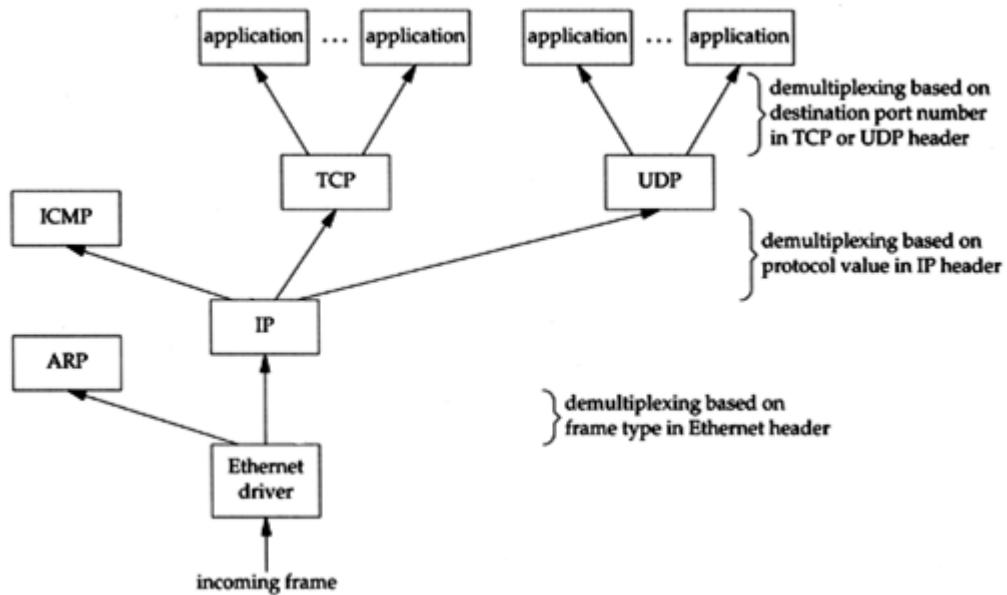


Figure 4. Demultiplexing of received ethernet frame (After Figure 1.8 in [3])

### 3. Data Packet Formats

This section will examine some of the common data formats from the transport layer down to the network interface layer.

### a. TCP Segment

The TCP segment between the transport layer and network layer is encapsulated as an IP datagram as shown in Figure 5. The format of the TCP header is shown in Figure 6. Each TCP segment has a source and destination port number, which are used by the sending and receiving applications, respectively. An IP address combined with a port number forms a *socket*. Together with the source and destination IP addresses in the IP header, the source and destination port numbers form a socket pair. In a TCP connection, a socket pair defines the two endpoints of the connection [9].

Both the *sequence number* and the *acknowledgement number* ensure that data transferred during a TCP connection is accounted for. The sequence number is used to reference the first byte of data that is to be sent. A transmitting TCP module will send over the data, then the receiving TCP module will add the sequence number to the number of bytes received and will use the total as the acknowledgement number. The acknowledgement number is then sent back to the sender, which uses it to reference the first byte of data for the next data transmission. The acknowledgement number sent by a sender is the sequence number the sender expects to receive.

The *header length* in the TCP header shows the number of 32-bit words in the header. The minimum TCP header length is 20 bytes (i.e. minimum value = 5 words). If options are used, the maximum length is 60 bytes (maximum value = 15 words). There are six *flags* (1 bit each) used in the TCP header. They are as follows:

URG: Urgent pointer valid

ACK: Acknowledgement number valid

PSH: Push data

RST: Reset connection

SYN: Synchronize sequence numbers

FIN: Finish connection

A 16-bit window (called 16-bit *window size*) is used for flow control purposes. There may be errors during data transmission and to provide protection against

transmission errors, the TCP header includes a 16-bit *checksum*. The checksum covers both the TCP header and the TCP data. The receiving TCP will verify this checksum by comparing this value with the one calculated based on the received TCP segment. The urgent pointer field in the TCP header is used when the URG bit is set. It is used when there is a need for priority data transfer. This field stores the sequence number of the last byte of the urgent data.

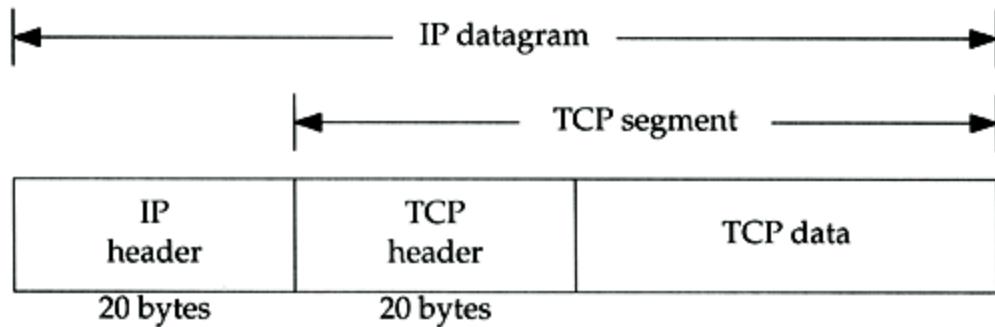


Figure 5. TCP segment in IP datagram (From Figure 17.1 in [3])

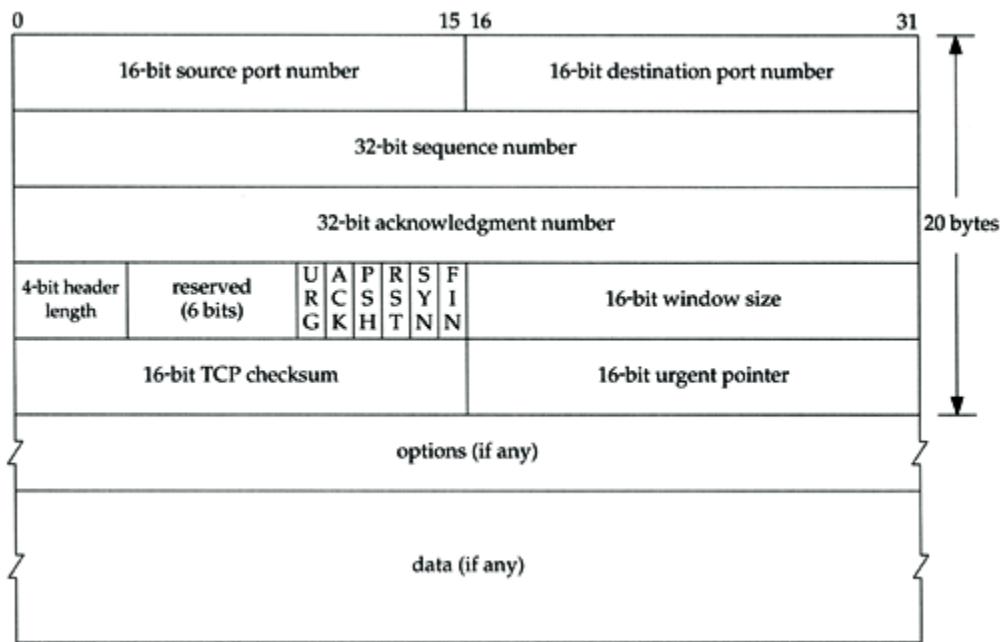


Figure 6. TCP header (From Figure 17.2 in [3])

### b. UDP Datagram

UDP is another transport layer protocol but is much simpler than TCP. A UDP datagram is encapsulated in an IP datagram as can be seen in Figure 7. Since UDP is a simple protocol, its header is only 8 bytes in length. Figure 8 shows the fields of a UDP header. The fields in the UDP header are comprised of source and destination port numbers, the total length of the UDP datagram (including header and data) and the checksum of the UDP datagram (including the UDP header and UDP data).

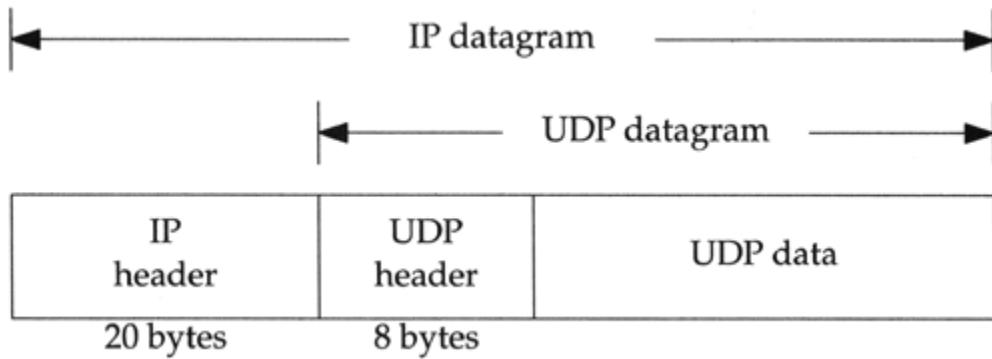


Figure 7. UDP datagram in IP datagram (From Figure 11.1 in [3])

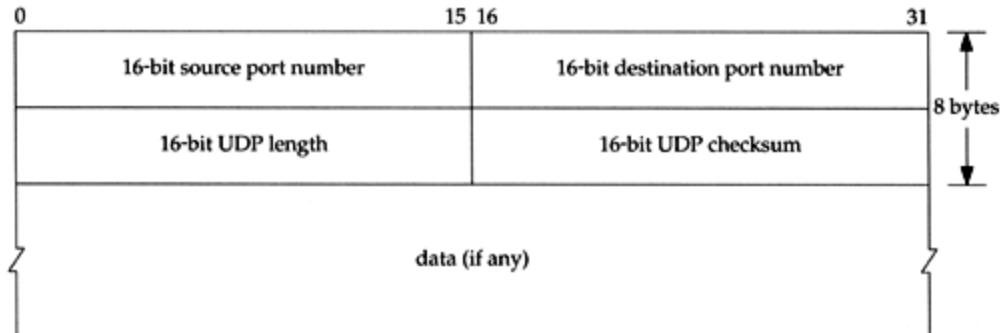


Figure 8. UDP header (From Figure 11.2 in [3])

### c. IP Datagram

TCP and UDP data eventually are transmitted as IP datagrams. Figure 9 shows the fields of an IP header. The *version* indicates the version of IP used. The *header length* denotes the length of the IP header including the options field. The *type of service*

(TOS) field is used in providing additional quality of services. Four types of services can be provided: minimize delay, maximize throughput, maximize reliability, and minimize cost. The *total length* field specifies the total length of the whole IP datagram. The *identification* field is used as an identifier of each datagram sent from a host. The 3-bit *flags* and 13-bit *fragmentation offset* are used during fragmentation. Of the 3-bit control flags, two are used to handle fragmentation: DF (Don't Fragment) and MF (More Fragment). The fragmentation offset specifies the position of original message that the fragmented data represents. *Time to live (TTL)* field denotes the number of remaining hops the datagram can be forwarded. The *protocol* field indicates which higher layer protocols are carried by the datagram. For example, some of the commonly used protocols are: TCP, UDP, ICMP, IGMP etc. The *checksum* field used for transmission error checking accounts for the IP header only. The IP header also has fields to specify the source IP address and destination IP address.

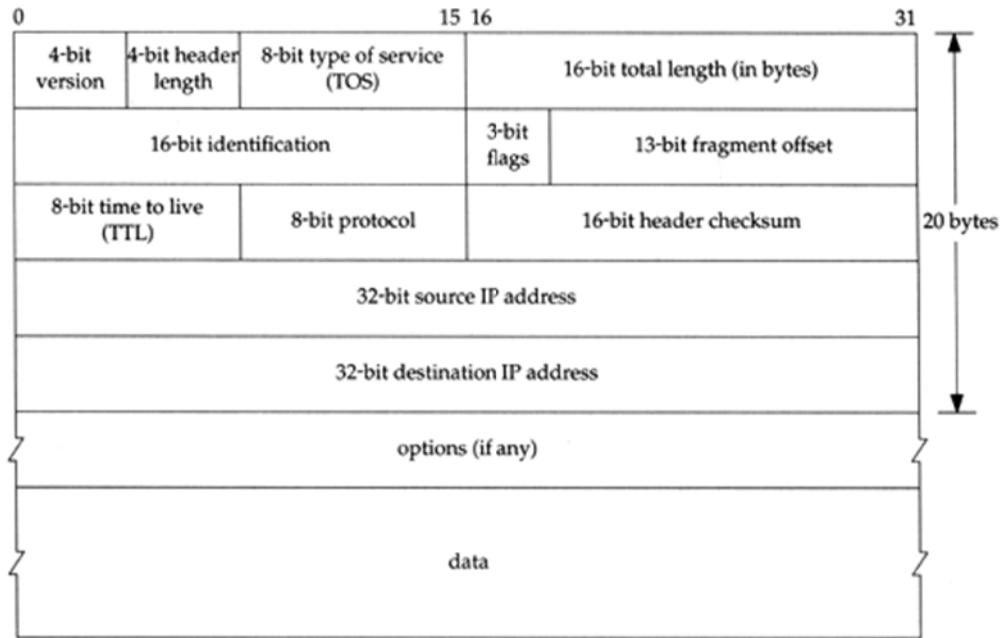


Figure 9. IP header (From Figure 3.1 in [3])

#### d. ICMP Message

ICMP messages are encapsulated in IP datagrams (see Figure 10). The format of an ICMP message is illustrated in Figure 11. The *type* field indicates which type of ICMP message the packet represents. For example, a *destination unreachable* ICMP message is of type 3, which indicates that the packet is undeliverable. The *code* field is used to provide specific details of the condition a particular ICMP message type is describing. For instance, an ICMP message of type 3 and code 3 indicates a “Destination Unreachable” error message. Code 3 implies that the port of the remote host is unreachable. Other ICMP message types and codes can be found in RFC792 [10]. The *checksum* field applies to the whole ICMP message including the header and data portion. The size of an ICMP message depends on the type and code of the ICMP message. Two additional fields (as can be seen in Figure 12): *identifier* and *sequence number* are required for Echo Request/Reply ICMP type message.

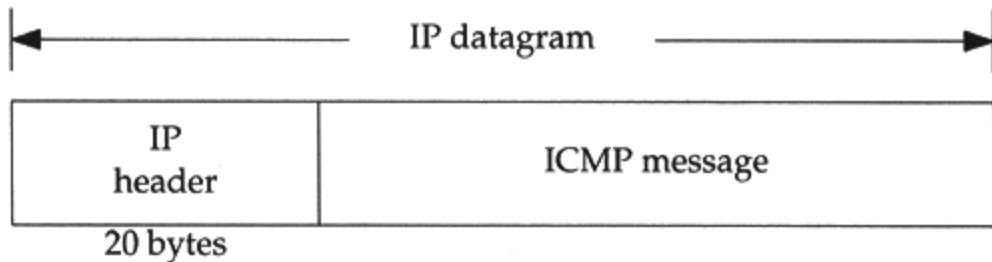


Figure 10. ICMP messages in IP datagram (From Figure 6.1 in [3])

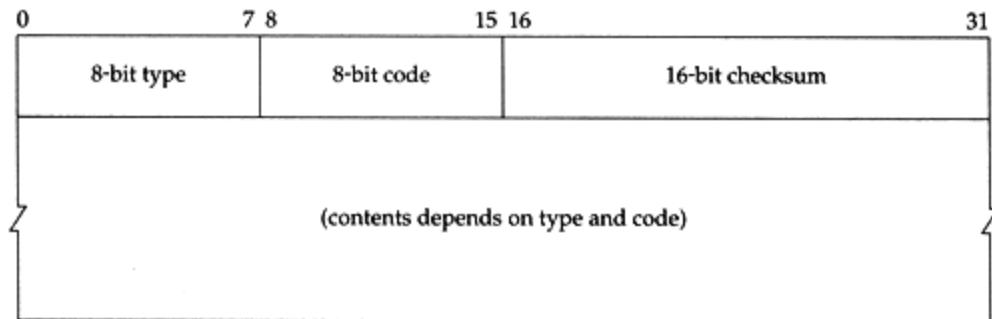


Figure 11. ICMP message (From Figure 6.2 in [3])

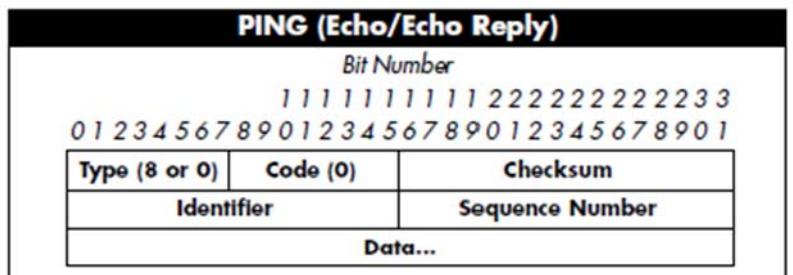


Figure 12. ICMP type message format for Echo Request and Reply (From [11])

### e. *Ethernet Frame*

The TCP/IP Protocol has support for different kinds of network interface layers, which depend on the type of hardware used. In our study, we will assume the use of Ethernet as the networking hardware. At the network interface layer, IP datagrams are encapsulated within Ethernet frames. The format of Ethernet encapsulation based on RFC 894 is shown in Figure 13. The Ethernet header consists of source and destination hardware addresses, a type field, and a cyclic redundancy check (CRC) field. Ethernet frames can be of three types: IP (0x0800), ARP (0x0806), or RARP (0x8035).

For the ARP type, the packet format for ARP requests and replies is shown in Figure 14. The *hardware address type* field indicates the type of hardware address (for example, Ethernet). The *protocol address type* field represents the type of protocol address being used (for example, IPv4). The *H/W Addr. Len* and *Prot. Addr Len* specify the number of bytes used by hardware addresses and protocol addresses, respectively. The *Operation* field denotes the type of operation: i.e., whether it is a Request or Reply operation. Other fields in the ARP packet are: source and destination hardware addresses, as well as source and destination protocol addresses.

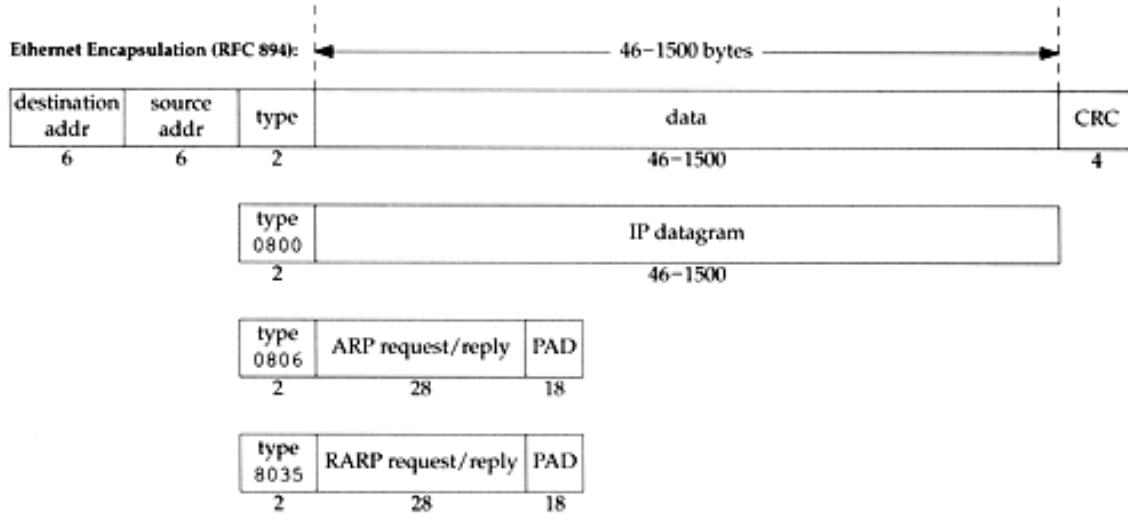


Figure 13. Ethernet encapsulation (From Figure 2.1 in [3])

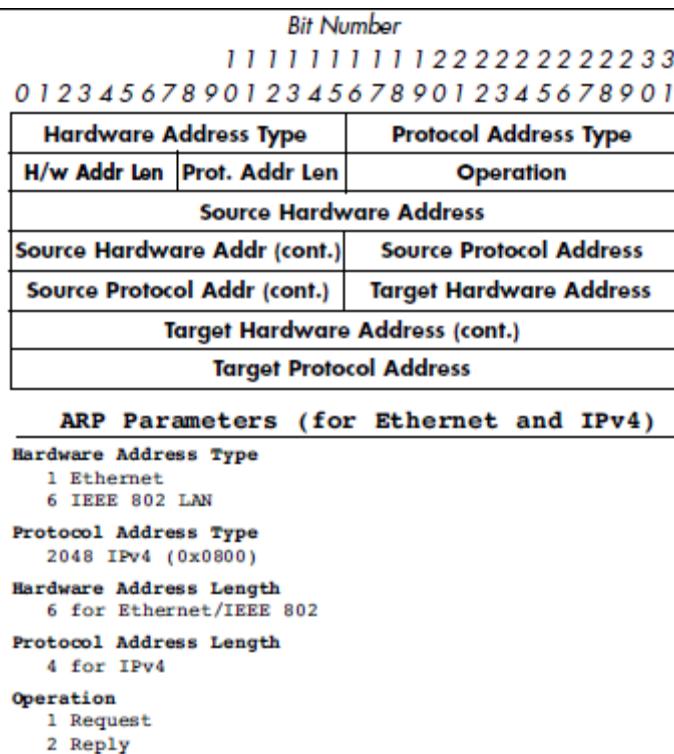


Figure 14. ARP Request/Reply packet format (From [11])

In summary, data packet formats are based on the Request for Comments (RFC) documents published by Internet Engineering Task Force (IETF) [12]. Interested readers can look up the following RFCs for detailed information on a particular data packet format.

- TCP – RFC793
- UDP – RFC768
- IP – RFC791
- ICMP – RFC792
- Ethernet – RFC894
- ARP – RFC826

## B. LEAST PRIVILEGE SEPARATION KERNEL

This section describes the Least Privilege Separation Kernel of the Trusted Computing Exemplar project. First, a brief review of separation kernels is provided.

### 1. Separation Kernels

A *kernel* forms the core component of an operating system. It manages system resources through inter-process communication mechanisms and system calls. User interactions with the system resources as well as data flows between user applications often depend on security policy enforcement by the kernel.

A special type of kernel is a *security kernel*. The set of security features for controlling access to system resources resides in the security kernel. A security kernel implements the reference monitor concept [13]; it is always invoked, tamperproof and small enough to be verifiable such that assurance for correctness and completeness is provided. A security kernel usually binds security labels to resources and then mediates access of subjects to resources based on the labels according to an internal security policy [14].

A *separation kernel* is a type of kernel introduced by Rushby in 1981 [15]. He noticed that, in a distributed system, data cannot flow between computers if they are not physically connected to one another. His idea for a separation kernel relies on the same principle. He proposed that, in a separation kernel, individual components of a system are separated into partitions, which mimic individual computers of a distributed system. Data can only flow between partitions if the partitions are “connected.” The connectivity between partitions is usually defined by a data flow policy. A separation kernel therefore allows components of a system to be separated into partitions and ensures that components within a partition cannot access information in another partition if they are not given permission to access that information.

Separation kernels are also known as partitioning kernels, as system resources are separated into partitions. Resources include subjects and objects. Separation kernels provide a resource-partition mapping to separate the resources among partitions. An inter-partition flow policy can be defined to specify the sharing of information between partitions. An example of inter-partition flow policy that can be enforced in a separation kernel is shown in Table 1. Subjects belonging to a particular partition can only access resources in another partition if the data flow policy enforced by the separation kernel allows it. Nevertheless, separation kernels only regulate data flow at the partition level. Finer granularity on the access permitted to subjects to interact with resources is usually not specified for separation kernels. Unlike typical separation kernels, a *least privilege separation kernel*, which will be discussed next, provides finer granularity regarding subject-to-resource accesses.

	<b>Partition A</b>	<b>Partition B</b>	<b>Partition C</b>
<b>Partition A</b>	RWX	W	-
<b>Partition B</b>	-	RWX	W
<b>Partition C</b>	-	-	RWX

Table 1. Inter-partition flow policy (From Table 1 in [16])

## 2. Least Privilege Separation Kernel

A Least Privilege Separation Kernel (LPSK) extends the separation kernel abstraction [16] with the *principle of least privilege* [17]. A separation kernel alone only defines a security policy on inter-partition flows. Resources in a partition are not visible to other partitions unless a flow is allowed between partitions. This means that if data flow is specified at the partition level, every subject in a particular partition can access any resource in another partition. In doing so, the principle of least privilege as required for high assurance systems is not realized. In contrast, an LPSK provides greater granularity on the interaction between subjects and resources within a partition. An LPSK extends the inter-partition flow concept of separation kernels with “subject-resource” flows. Given a data flow between two partitions as specified by a separation kernel, an LPSK also defines how each subject of a partition is allowed to access resources either in its own partition or in other partitions. For example, a least privilege separation kernel configuration is illustrated in Figure 15. The arrows in the illustration represent the data flow to or from a subject and a resource.

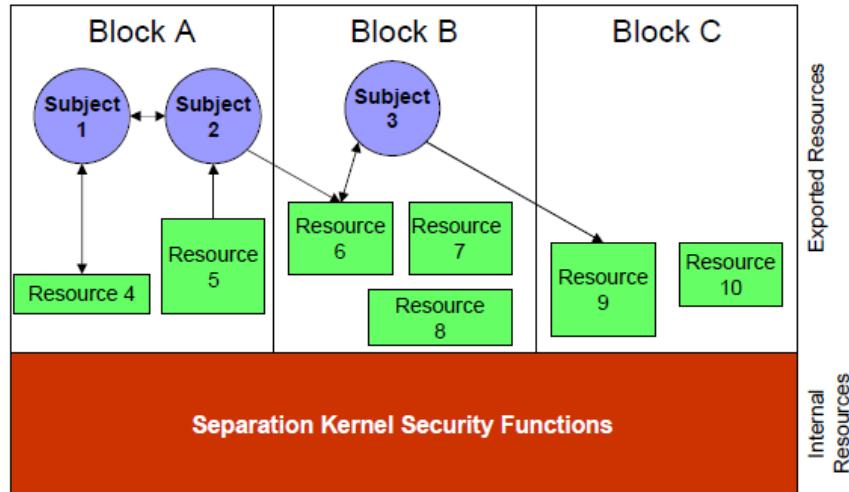


Figure 15. LPSK configuration (From Figure 1 in [16])

In Figure 15, there are three partitions: A, B and C. Partition A consists of two subjects: Subject 1 and Subject 2. Subject 1 can read and write to Resource 4 whilst

Subject 2 can only read from Resource 5 and write to Resource 6, which is in another partition B. Subject 3 in partition B can read from and write to Resource 6 as well as write to Resource 9, which belongs to partition C. Subject 3 has no access to Resources 7 and 8 in its own partition B. The data flow between subjects and resources can be defined in the form of a matrix as shown in Table 2.

		Resources					
		1	2	4	5	6	9
Subjects	1	-	RW	RW	-	-	-
	2	RW	-	-	R	W	-
	3	-	-	-	-	RW	W

Table 2. Subject-Resource flow matrix (From Table 3 in [16])

### 3. Trusted Computing Exemplar Project

The main purpose of the Trusted Computing Exemplar (TCX) Project is to provide a “worked example of how high assurance trusted computing components can be built” [2]. The TCX project has four deliverables:

- Creation of a prototype framework for rapid high assurance system development
- Development of a reference implementation trusted computing component
- Evaluation of the component for high assurance
- Open dissemination of deliverables related to the first three activities [1].

The trusted computing component implemented in TCX project is the LPSK kernel, the implementation of which is in compliance with U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness [18].

**THIS PAGE INTENTIONALLY LEFT BLANK**

### **III. REQUIREMENTS AND DESIGN**

This chapter first explains the requirements for the implementation of an IP protocol stack in the LPSK, and then discusses the design considerations for the IP protocol stack implementation. Section A describes the functional requirements for the system prototype as well as for the IP protocol stack. Section B describes a high level design of the IP protocol stack followed by an explanation of the placement of the IP protocol stack in the LPSK. Section B also reviews a few open source network protocol stacks from which the Lightweight Internet Protocol (LWIP) TCP/IP stack eventually is selected. Some of the relevant LWIP data structures and functions are highlighted in the same section. The section finishes with an elaboration of the final design of an IP protocol stack.

#### **A. REQUIREMENTS**

The current implementation of the LPSK lacks a hardware driver module for networking devices. An interim solution prior to the development of a complete hardware driver is to develop a network packet generator from which the network protocol stack can consume generated packets. The requirements of this project thus can be separated into functional requirements for the primitive system prototype to demonstrate networking capability and functional requirements for the network protocol stack. As the main objective of this project is to demonstrate the concepts of IP networking in the LPSK prototype, the requirements for the full suite of the TCP/IP protocol stack will not be considered. Instead, only the IP protocol stack will be considered in the preliminary demonstration of networking functionality in the LPSK.

##### **1. Functional Requirements for the System Prototype**

Table 3 provides a list of functional system requirements (SR) for the system prototype. The set of requirements ensure that the working prototype is able to generate and handle IP packets. In order to achieve such requirements, an IP protocol stack (SR1-3 and SR2-3) must be in place. The requirements for the IP protocol stack not only must be able to encapsulate transport layer data and demultiplex Ethernet data, it must also be

able to provide other functions. The set of functional requirements for an IP protocol stack are listed in Table 4 and will be explained next.

<b>Requirement No.</b>	<b>Functional Requirements for System Prototype</b>
<b>SR1</b>	<b>Generation of IP packets</b>
SR1-1	The system shall allow users to generate IP packets
SR1-2	The system shall allow users to send IP packets to remote hosts
SR1-3	The system shall provide an IP protocol stack to encapsulate data in an Ethernet frame
<b>SR2</b>	<b>Handling of IP packets</b>
SR2-1	The system shall allow users to receive IP packets from remote hosts
SR2-2	The system shall process received IP packets
SR2-3	The system shall provide an IP protocol stack to demultiplex data in an Ethernet frame

Table 3. Functional requirements for the system prototype

## 2. Functional Requirements for the IP Protocol Stack

The high level functional requirements (FR) for the IP protocol stack are shown in Table 4. Functions for the IP protocol stack can be divided into two groups: (FR1) those providing services to other protocols and (FR2) those providing services to end-users [19]. However, the IP protocol stack will not include functionality to support the full TCP/IP protocol suite. It will only focus on Network layer and Ethernet layer functionality. FR1 can be subcategorized as FR1-1 and FR1-2 to represent Network layer services and Ethernet layer services, respectively. As for FR2, there are many useful end users services, however, since the project only extends up to the network layer in the protocol stack, the applicable services are limited to those that do not require the transport layer protocol. A *ping* application is a suitable choice because it does not require TCP or UDP of the transport layer. However ping requires ICMP, which is part of the network layer. The requirement for the IP protocol stack to support ping is denoted by FR2-1.

<b>Requirement No.</b>	<b>Functional Requirements for the IP Protocol Stack</b>
FR1	IP protocol stack shall provide services to the Network protocol layer and the Ethernet protocol layer
FR1-1	IP protocol stack shall provide Network layer services
FR1-2	IP protocol stack shall provide Ethernet layer services
FR2	IP protocol stack shall provide services to end users
FR2-1	IP protocol stack shall support the ping application

Table 4. Functional requirements for the IP protocol stack

The functional requirements of the network layer services provided by the IP protocol stack can be further expanded into requirements FR1-1-1 to FR1-1-4. They are listed in Table 5. The protocols required in the network layer are Internet Protocol (IP) and Internet Control Message Protocol (ICMP). Only support for IP version 4 (IPv4) and ICMP version 4 (ICMPv4) are considered. Formatting of data encapsulation for IP datagrams is supported in the network layer. The format of the IP header and the IP datagram is based on the formats shown in Figure 9 and Figure 3 (see Chapter II). The routing functionality provided by the protocol stack will select the network interface that is on the same network as the destination host in a networked environment.

<b>Requirement No.</b>	<b>Functional Requirements for Network Layer Services (FR1-1)</b>
FR1-1-1	IP protocol stack shall support IP addressing (IPv4)
FR1-1-2	IP protocol stack shall support IP datagram data encapsulation and formatting
FR1-1-3	IP protocol stack shall provide routing services
FR1-1-4	IP protocol stack shall support Internet Control Message Protocol (ICMP) version 4

Table 5. Functional requirements for network layer services (FR1-1)

For Ethernet layer services, the expanded set of functional requirements to be provided by the IP protocol stack are listed in Table 6. Address Resolution Protocol

(ARP) is supported by the IP protocol stack. The format of encapsulation of information into Ethernet frames shall follow the requirements in RFC894 as shown in Figure 13 (see Chapter II).

<b>Requirement No.</b>	<b>Functional Requirements for Ethernet Layer Services (FR1-2)</b>
FR1-2-1	IP protocol stack shall provide services for Address Resolution Protocol (ARP)
FR1-2-1-1	IP protocol stack shall allow processes to send ARP request and reply packets
FR1-2-1-2	IP protocol stack shall allow processes to receive ARP request and reply packets
FR1-2-1-3	IP protocol stack shall allow processes to update the local ARP cache
FR1-2-2	IP protocol stack shall support Ethernet Encapsulation (RFC 894)

Table 6. Functional requirements for ethernet layer services (FR1-2)

The IP protocol stack shall support the *ping* application (See Table 4). The requirements for the IP protocol stack to handle the *ping* application are listed in Table 7. For the *ping* application to work successfully, it must be able to ping a remote host and receive ping replies from that host. The IP protocol stack shall be able to process the ping packets and send appropriate ping replies to the originating host.

<b>Requirement No.</b>	<b>Functional Requirements for Ping Application (FR2-1)</b>
FR2-1-1	IP protocol stack shall allow users to ping remote hosts
FR2-1-2	IP protocol stack shall process ping packets
FR2-1-3	IP protocol stack shall allow users to receive ping replies from remote hosts

Table 7. Functional requirements for ping application (FR2-1)

## B. DESIGN CONSIDERATIONS

This section first describes a high level design for the IP protocol stack. Some design considerations such as how the protocol stack should be organized in the LPSK, and the layering design methodology used in the LPSK are also discussed. A comparison of two open source TCP/IP protocol stacks, namely MicroIP (uIP) and Lightweight IP (LWIP), is provided. In particular, LWIP is selected over uIP and some of LWIP's relevant data structures and function calls are highlighted. Lastly, the final design with some modifications to the LWIP design is explained.

### 1. High Level IP Protocol Stack Design

From the functional requirements described in Section A, a high level design of the IP protocol stack can be envisaged in Figure 16. The IP protocol stack can be designed so that each box represents a module and each arrow denotes the interface necessary between modules. For example, the Ethernet layer consists of the ARP module and the Ethernet module. There will be an interface between the ARP and Ethernet module within the Ethernet layer. Outside the Ethernet layer, there will be an interface between IP module in the Network layer and the Ethernet layer. For the ping application to work, it does not require protocols in the transport layer. The transport protocols: TCP and UDP illustrated in the figure show that there should be an interface between the Network layer and Transport layer, but this is beyond the scope of the current project. The TCP and UDP modules and their interfaces between other modules and layers will not be included in the detailed design.

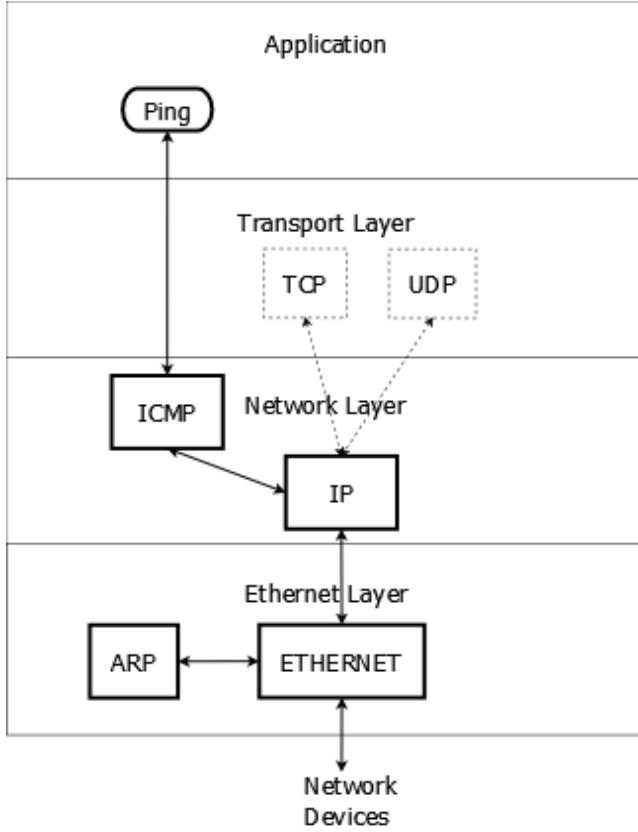


Figure 16. High level IP protocol stack design

## 2. IP Protocol Stack in the TCX LPSK

The current implementation of the TCX LPSK has the architecture shown in Figure 17. The functions pertaining to each layer can be found in LPSK Functional Requirement Specifications [20]. Since the overall objective of this project is to develop a proof-of-concept prototype involving the use of an IP protocol stack, the protocol stack can reside in any non kernel privilege levels (PL1 through PL3). To simplify the design, the IP protocol stack will be implemented in PL3.

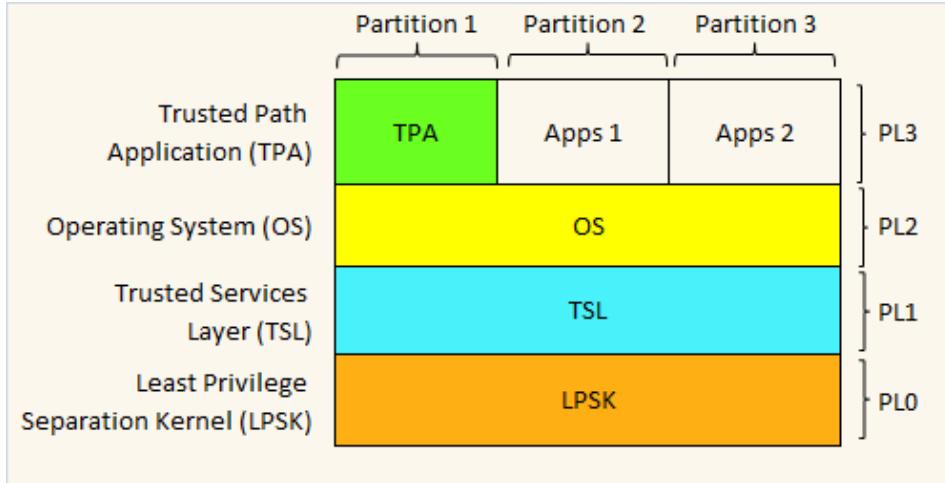


Figure 17. Architecture for the TCX LPSK (After Figure 1 in [21])

### 3. Layering in the TCX LPSK

The design methodology adopted by the TCX LPSK uses modularity and layering. This organization is illustrated notionally in Figure 18. The LPSK is decomposed into modules and the modules are organized into ordered layers. Modules belonging to upper layers can only make calls into modules in the lower layers. It is important to note the usefulness of modularity in secure system design, as mentioned by Levin et al. [22]. Therefore, for the design of the IP protocol stack, the concept of layering will be applied to the modules in the network stack.

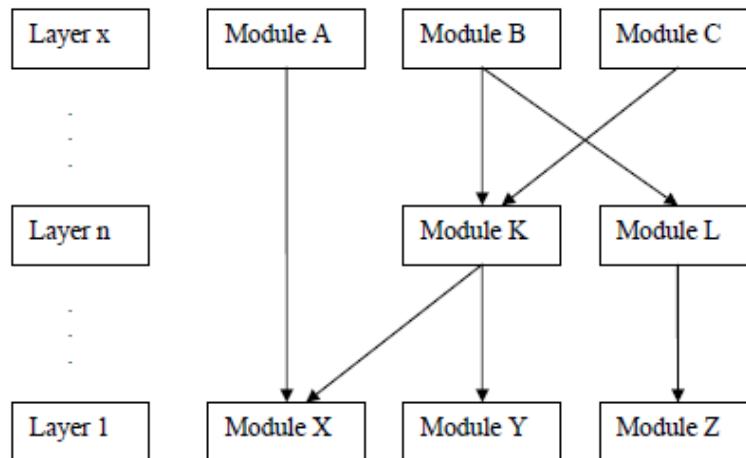


Figure 18. Modules and layering in the TCX LPSK

#### **4. Open Source Network Protocol Stacks**

To develop a network IP protocol stack, one can either build everything from scratch or search for a suitable product that can be adapted to match the requirements. Open source is a better consideration over proprietary off-the-shelf software solutions due to the fact that source code is usually released for open source implementations. Being able to access the source code allows modifications to be made such that there is flexibility in customizing existing code to better suit the requirements. The approach taken in this project is to examine relevant software in the open source community. Fortunately, there are various software network protocol stacks available as open-source implementations.

The four open-source network protocol stacks evaluated are microIP (uIP) [23], Lightweight IP (LWIP) [24], TinyTCP [25], uC/IP [26]. The protocol stacks were evaluated based on the criteria listed in Table 8. The minimum requirements that need to be satisfied are that the protocol stack must be able to run in a Linux environment and support IP, ICMP, ARP, Ethernet and IPv4. Only uIP, LWIP and uC/IP are able to meet the minimum requirements. Other selection criteria include availability of source code, documentation, an active user community and sample applications, the ability to support IPv6, as well as the TCP and UDP protocols. Although uC/IP meets the minimum requirement, it fails to have any documentation of its source code. uC/IP is unable to support IPv6 and does not have many active users. Out of the four network protocol stacks, it is clear that uIP and LWIP are the two most suitable protocol stacks to be used in this project.

	<b>uIP</b>	<b>LWIP</b>	<b>TinyTCP</b>	<b>uC/IP</b>
Supports TCP	✓	✓	✓	✓
Supports UDP	✗	✓	✗	✓
Supports ICMP	✓	✓	✗	✓
Supports IP	✓	✓	✓	✓
Supports ARP	✓	✓	✓	✓
Supports Ethernet	✓	✓	✓	✓
Support IPv4	✓	✓	✓	✓
Support IPv6	✓	✓	✗	✗
Supports Linux environment	✓	✓	✓	✓
APIs / Source codes available	✓	✓	✓	✓
Documentation available	✓	✓	✗	✗
Active users community	✓	✓	✗	✗
Include sample applications	✓	✓	Few	Few
Latest version no.	1.0.0	1.4.0	2	1.0.3
Last version date (mm-yy)	Jun-07	Jul-10	Sep-97	Jan-02

Table 8. Evaluation criteria of network protocol stacks

Further comparisons of uIP and LWIP are listed in Table 9. LWIP is a better choice over uIP, as it has other features that provide leeway for extending the capability of this project. For instance, LWIP supports the UDP protocol and multiple network interfaces whereas uIP does not. Moreover, TCP features like TCP sliding window, congestion control, out-of-sequence data are supported in LWIP but not in uIP. If future expansion of this project is to include the TCP protocol, LWIP is truly the best ultimate choice. In addition, LWIP is selected on the merit that it is still under active development. The latest update for LWIP was released in Jul 2010. As for uIP, the last update was in Jun 2007. Although LWIP requires greater amount of memory to work with as compared with uIP, memory is not an issue, as existing systems running the LPSK have 512 MB of memory, which is more than sufficient for LWIP. Further analysis of the LWIP architecture is necessary to determine if it can fulfill the requirements and if it can be integrated into the TCX LPSK. This appears in the next section.

<b>Feature</b>	<b>uIP</b>	<b>LWIP</b>
IP and TCP checksums	✓	✓
IP fragment reassembly	✓	✓
IP options		
Multiple interfaces		✓
UDP		✓
Multiple TCP connections	✓	✓
TCP options	✓	✓
Variable TCP MSS	✓	✓
RTT estimation	✓	✓
TCP flow control	✓	✓
Sliding TCP window		✓
TCP congestion control	Not needed	✓
Out-of-sequence TCP data		✓
TCP urgent data	✓	✓
Data buffered for rexmit		✓

Table 9. TCP/IP features implemented by uIP and LWIP (From Table 1 in [27])

## 5. Lightweight Internet Protocol (LWIP)

The design of LWIP is based on the layered characteristics of each protocol in the TCP/IP suite. Each protocol is implemented separately as a module, and has a few functions to provide interfaces to other protocols. Nevertheless, Dunkels stated that the main purpose of implementing LWIP is to “reduce memory usage and code size”, and “improve performance in terms of processing speed and memory usage” [27]. If the protocols are implemented in a strictly layered manner, there will be communication overhead, which will degrade the overall performance [28]. As such, Dunkels claimed that LWIP is designed to have “a more relaxed scheme for communication” between layers through the use of its “buffer handling mechanisms” [27]. Some of LWIP’s data structures and functions are relevant to the requirements of this thesis. They are identified and discussed in the next section.

## 6. LWIP Data Structures

Instead of building from scratch, some of the data structures in LWIP can be reused to achieve our objectives. The data structures are as follows:

### a. *Packet Buffer (pbuf)*

LWIP uses a packet buffer (pbuf) to represent a packet. The description of each field of the pbuf's data structure is shown in Table 10. There are three types of pbuf: PBUF\_RAM, PBUF\_ROM, and PBUF\_POOL [27]. For PBUF\_RAM pbuf, memory is allocated to store application data as well as header information of IP packets. It is commonly used by applications that send dynamically generated data. PBUF\_ROM is used when the application data is stored in a memory location managed by the application. The PBUF\_POOL pbuf is allocated from a pool of fixed size pbufs. It is mainly used by network device drivers due to its fast operation. Pbufs can be linked together in a list and a chain of pbufs can be comprised of different types of pbufs.

pbuf		
Type	Name	Description
struct pbuf*	next	Next pbuf of a linked pbuf chain
void *	payload	Actual data in the buffer
unsigned short	tot_len	Total length of this buffer and all next buffers in the chain
unsigned short	len	Length of this buffer
unsigned char	type	Buffer type
unsigned char	flags	Miscellaneous flags
unsigned short	ref	No. of pointers that reference this pbuf

Table 10. Packet buffer data structure (From [27])

### b. Network Interface (netif)

LWIP represents device drivers for network hardware using a network interface (netif) structure [27]. Descriptions of the fields in the netif data structure are given in Table 11. Network interfaces are globally stored in a linked list. Each netif has three important functions: input, output, linkoutput. The *input* function is called when a packet is received. It is used to pass a packet from the Ethernet layer to the upper layer. The *output* function sits between the network layer and the network interface layer. When sending a packet to a remote host, the *output* function is called by the IP module in order for the packet to traverse from the network layer to the network interface layer. The *linkoutput* function is an interface between the network interface layer and the physical device. It is called by the ARP module to send a packet to the physical device.

netif		
Type	Name	Description
struct netif*	next	Next netif in linked list
ip_addr_t	ip_addr	IP address of network interface
ip_addr_t	netmask	Network address of network interface
ip_addr_t	gw	Gateway
netif_input_fn	input	input function
netif_output_fn	output	output function
netif_linkoutput_fn	linkoutput	output function on link medium
void*	state	state information for device
unsigned short	mtu	Max transfer unit (in bytes)
unsigned char	hwaddr_len	No. of bytes used in hardware address
unsigned char[HWADDR_LEN]	hwaddr	Hardware address of this interface
unsigned char	flags	Flags for this interface
char [2]	name	Name of interface (eg. et)
unsigned char	num	No. of this interface

Table 11. Network interface data structure (From [29])

*c. ICMP Echo Header (icmp\_echo\_hdr)*

The ICMP echo header data structure is used to represent the ICMP header information described in Figure 12 (see Chapter II). The details of the fields of the data structure are listed in Table 12.

icmp_echo_hdr		
Type	Name	Description
unsigned char	type	ICMP type
unsigned char	code	ICMP code
unsigned short	chksum	Checksum
unsigned short	id	Identifier
unsigned short	seqno	Sequence number

Table 12. ICMP echo header data structure (From source code in [30])

*d. IP Header (ip\_hdr)*

The details of the IP header data structure are shown in Table 13. The data structure is used to represent IP header information shown in Figure 9 (see Chapter II).

ip_hdr		
Type	Name	Description
unsigned char	v_hl_tos	Version / Header Length / Type of service
unsigned short	len	Total length
unsigned short	id	Identification
unsigned short	offset	Fragmentation offset
unsigned char	ttl	Time to live
unsigned char	proto	Protocol type
unsigned short	chksum	Checksum
ip_addr_t	src	Source IP address
ip_addr_t	dest	Destination IP address

Table 13. IP header data structure (From source code in [30])

#### e. *Ethernet Header (eth\_hdr)*

The Ethernet header (eth\_hdr) data structure represents the Ethernet header information described in Figure 13 (see Chapter II). The description of the data structure is listed in Table 14.

eth_hdr		
Type	Name	Description
eth_addr	dest	Destination Ethernet Address
eth_addr	src	Source Ethernet Address
unsigned short	type	Ethernet type

Table 14. Ethernet header data structure (From source code in [30])

#### *f. ARP Header (etharp\_hdr)*

The ARP header (etharp\_hdr) data structure represents ARP packet format information shown in Figure 14 (see Chapter II). The details of the data structure are described in Table 15.

etharp_hdr		
Type	Name	Description
unsigned short	hwtype	Hardware type
unsigned short	proto	Protocol type
unsigned char	hwlen	Hardware address length
unsigned char	protolen	Protocol address length
unsigned short	opcode	Operation code
eth_addr	shwaddr	Source hardware address
ip_addr_t	sipaddr	Source IP address
eth_addr	dhwaddr	Destination hardware address
ip_addr_t	dipaddr	Destination IP address

Table 15. ARP header data structure (From source code in [30])

## 7. LWIP Functions and Function Call Flow

The LWIP functions to be used in this project are summarized in Table 16. For IP Processing, the IP module implemented in LWIP can only send, receive and forward packets. It does not have the capability to process fragmented packets and IP options. Sending packets is handled by the *ip\_output()* function. The function finds the appropriate network interface (using *ip\_route()* ), determines the source and destination IP address, and subsequently calls *ip\_output\_if()* to construct the IP header and send the packet on the network interface using the *netif->output()* function call. The *netif->output()* function can be assigned to call *etharp\_output()* of ARP module. The

*etharp\_output()* function resolves and fills in the Ethernet address of the outgoing IP packet, and eventually sends the IP packet through the *netif->linkoutput()* function.

Incoming packets are handled by the *netif->input()* function. The *netif->input()* function can be assigned to call *ethernet\_input()* function. The *ethernet\_input()* function processes incoming Ethernet frames. Depending on the type of Ethernet frame it receives, the function will call *ip\_input()* if the Ethernet frame is an IP type frame, or it calls *etharp\_arp\_input()* if the Ethernet frame is of the ARP type. The function *etharp\_arp\_input()* handles ARP requests and ARP replies. If an ARP request is received, it responds with an ARP reply. If an ARP reply is received, it updates the ARP cache. Upon receiving a packet, the *ip\_input()* function will make a function call to an appropriate protocol in the upper layer, depending on the protocol type of the IP datagram. If the IP protocol is of an ICMP type, then *ip\_input()* will call the function *icmp\_input()*. The function *icmp\_input()* handles incoming ICMP packets. The current implementation can only process ICMP echo requests and send out echo replies.

Function Name	Description
ip_output( )	Find appropriate network interface Ensure that all IP header fields are filled Construct IP header and compute IP header checksum Determine source and destination IP addresses
ip_input( )	Check IP version, header length Compute header checksum Check destination IP address
ip_route( )	Find appropriate network interface for a given IP address
ip_output_if()	Sends IP packet on a network interface
ip_forward( )	Decrease TTL field If TTL = 0, send ICMP error message
etharp_output( )	Resolve and fills in Ethernet address header for outgoing IP packets
ethernet_input( )	Processes received Ethernet frames. Passes frame to ARP module if it receives ARP frame Passes frame to IP module if it receives IP ethernet frame
etharp_arp_input( )	Responds to ARP requests Updates ARP cache if it receives ARP replies
icmp_input( )	Process icmp echo request Sends out echo response
etharp_request( )	Sends ARP request
etharp_send_ip()	Sends IP packet on a network using netif->linkoutput.

Table 16. LWIP functions (From [24])

The flow of function calls for sending and receiving IP packets in LWIP can be best illustrated with a diagram shown in Figure 19. The sequence of function calls when sending IP packets is marked from 1a to 1f. For incoming packets, the sequence of function calls is indicated from 2a to 2b. Step 0 denotes a user process or user application. For instance, when a user application such as ping sends a ping request to a remote host, the data is first handled by the *raw\_sendto()* function. It calls *ip\_route()* to determine the network interface based on the destination IP address, and then calls the *ip\_output\_if()* function to send the IP packet on the network interface. The *ip\_output\_if()* function next calls the *netif->output()* function, which finally calls *netif->linkoutput()* to send the IP packet through the link. On receiving incoming packets, the *netif->input()*

function determines whether to call *etharp\_arp\_input()* or *ip\_input()*, based on the type of ethernet frame received. If an Ethernet IP frame is received, then the *ip\_input()* function will make an up call to *icmp\_input()* to process ICMP packet. If the *icmp\_input()* function receives an ICMP *echo reply*, it may signal to the application by calling a function in the user application. The flow sequence for sending and receiving packets is summarized in Figure 20 and Table 17.

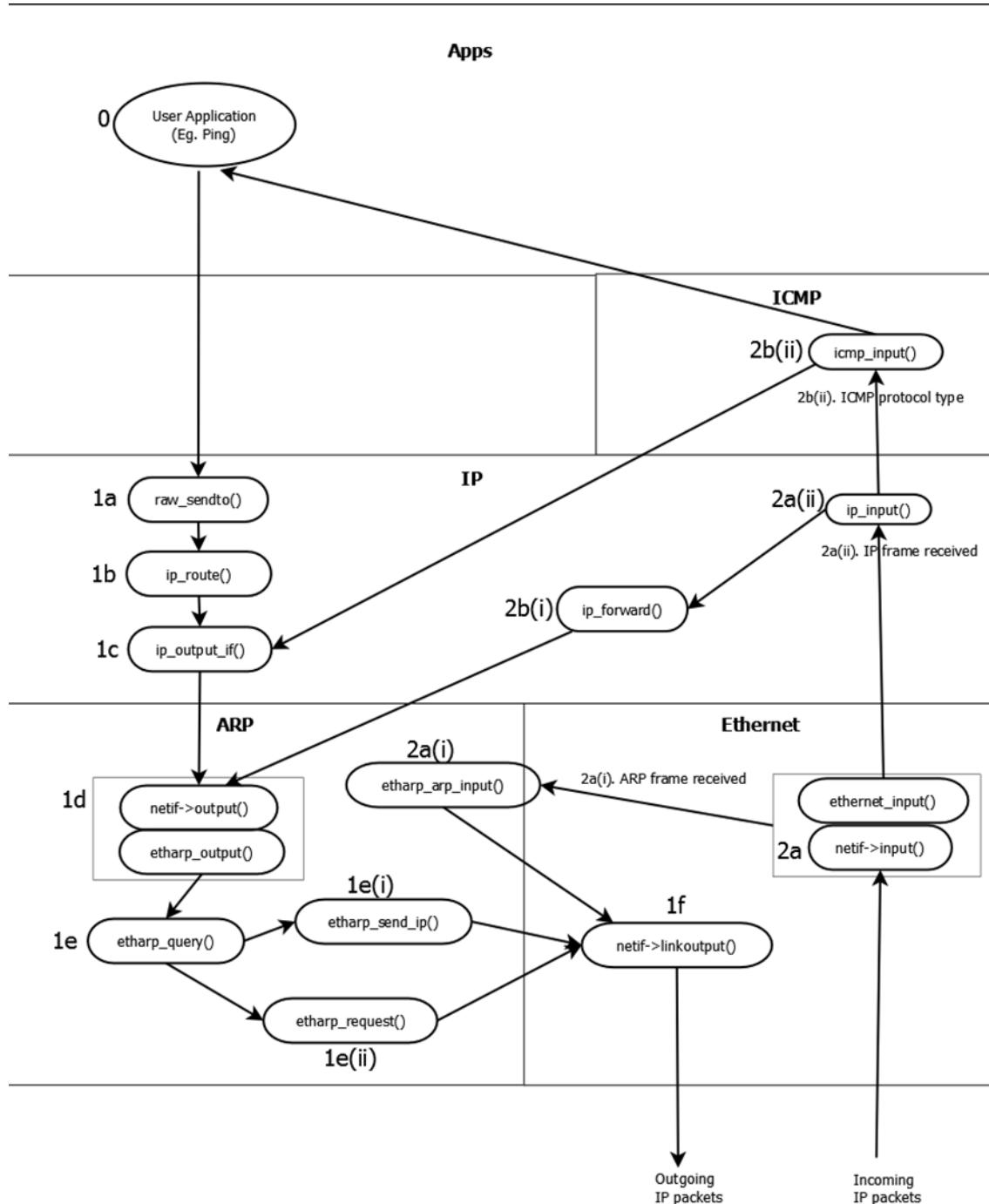


Figure 19. LWIP function call flow for sending and receiving IP packets

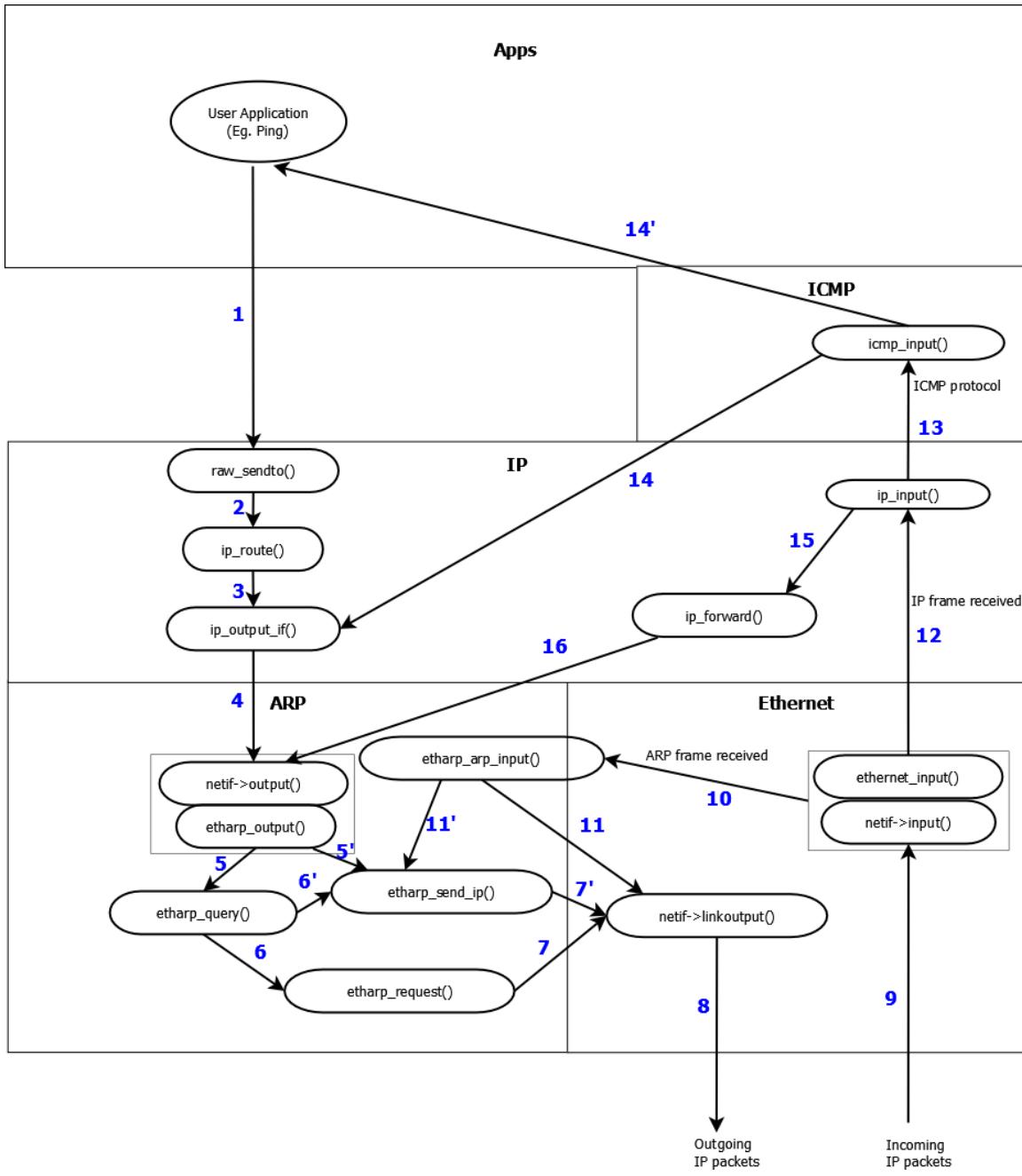


Figure 20. Flow sequence for sending and receiving IP packets

Action	Scenario	Flow Sequence
Sending Packet	Destination MAC address is not in ARP cache	1-2-3-4-5-6-7-8
Sending Packet	Destination MAC address is in ARP cache	1-2-3-4-5-6'-7'-8
Receiving Packet	Incoming packet is ARP request packet	9-10-11-8
Receiving Packet	Incoming packet is ARP reply packet	9-10-11'-7'-8
Receiving Packet	Incoming packet is ICMP Echo packet	9-12-13-14-4-5'-7'-8
Receiving Packet	Incoming packet is ICMP Echo Reply packet	9-12-13-14'
Receiving Packet	Incoming packet is not for host, MAC address of host not in ARP cache	9-12-15-16-5-6-7-8
Receiving Packet	Incoming packet is not for host, MAC address of host in ARP cache	9-12-15-16-5-6'-7'-8

Table 17. Flow sequence for sending and receiving IP packets

It can be observed from Figure 19 that sending of IP packets involves downward function calls and receiving of IP packets required upward function calls in LWIP. However, the upward function calls when receiving IP packets are not desired if LWIP is to be integrated with LPSK. This is because LPSK was designed such that modules can only make downward function calls to other modules. Therefore, in order to allow LWIP to fulfill design criteria of the larger LPSK effort, modification of the LWIP architecture is necessary especially for the sequence in which input functions call each other. In other words, input functions should only make downward calls to other functions. The final design with modifications to LWIP together with interface to LPSK functions will be discussed next.

## 8. Final Design

The final design of the IP protocol stack is shown in Figure 21. The *lpsk\_write\_next()* and *lpsk\_read\_next()* functions (represented by rectangle boxes at the bottom of the figure) are implemented in the current LPSK to write bytes to network devices and read bytes from network devices, respectively. Incoming Ethernet frames are stored in a buffer, which can be polled in a first-in-first-out manner by calling *lpsk\_read\_next()* function. Outgoing Ethernet frames can be sent to the network device using *lpsk\_write\_next()*.

One of the major changes between the final design and the original LWIP design is the flow of function calls for receiving frames. Instead of the lower layer functions calling upper layer functions as received frames are processed, the modified design ensures that functions in the upper layers call functions in the lower layers. As an example, when ping waits for a ping reply, it calls the *recv\_ping\_reply()* function to get an IP packet, which is the ping reply. To be precise, the IP packet uses ICMP protocol and the ICMP type equals Echo Reply (Type 0). Since the ping reply is an ICMP message, the *recv\_ping\_reply()* function will subsequently call the *get\_icmp\_packet()* function to get an IP datagram having the ICMP protocol. As such, the *get\_icmp\_packet()* function calls the *get\_ip\_datagram()* function to get an IP datagram of ICMP protocol type.

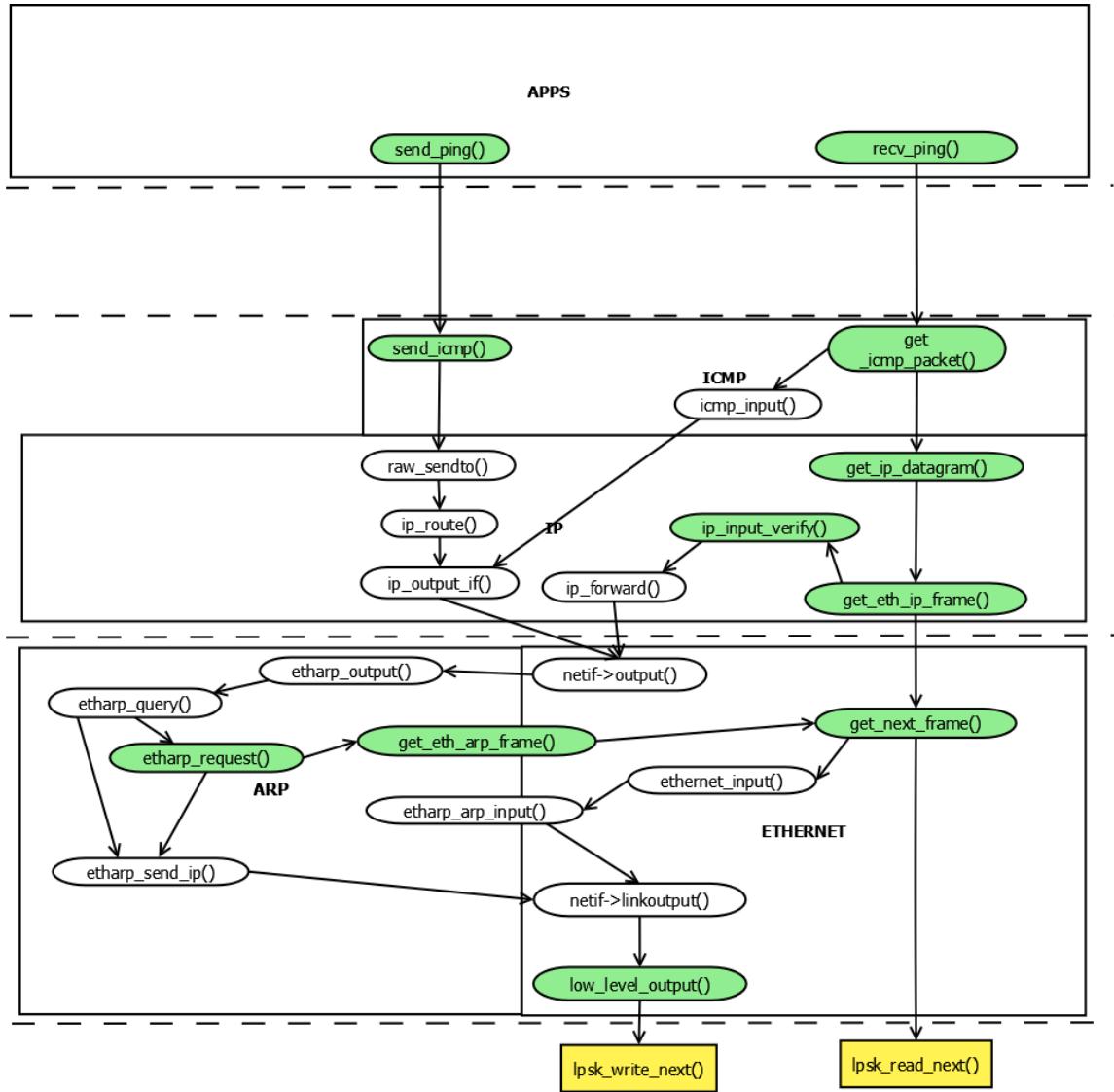


Figure 21. IP protocol stack final design

The `get_ip_datagram()` function stores the IP datagram in a `ip_datagram_buffer` data structure shown in Table 18. The data structure is designed to store IP datagrams of various protocol types. An `ip_datagram_buffer` list will be globally defined for storing the received IP datagram. The `get_ip_datagram()` function then calls the `get_eth_ip_frame()` function to get an Ethernet frame, which is of IP type. When the `get_eth_ip_frame()` function gets an Ethernet-IP frame, it will then call `ip_input_verify()` function to check and verify the IP version, header length and checksum of the packet. The

*ip\_input\_verify()* function is a modification of LWIP's *ip\_input()* function such that it will not make an up call to functions handling ICMP messages. If the destination IP address does not match the IP address of receiving host, the packet will be forwarded to the network interface, which belongs to the same network as the destination IP address using *ip\_forward()* function. In order to get an Ethernet frame, the *get\_eth\_ip\_frame()* function calls *get\_next\_frame()*, which eventually calls *lpsk\_read\_next()* to get an incoming Ethernet frame.

ip_datagram_buffer		
Type	Name	Description
struct ip_datagram_buffer *	next	Pointer to next ip_datagram_buffer of a linked list
struct pbuf*	p	Packet buffer
unsigned short	ip_proto_type	IP Protocol Type
unsigned int	len	Length of IP datagram
char [4]	netif_name	Name of netif eg. 'eth1'

Table 18. Data structure for IP datagram buffer

When the *get\_next\_frame()* function is called, it will first get all the frames received by the network device and store the frames in its frame buffer (eth\_frame\_buffer). The eth\_frame\_buffer data structure for storing the incoming Ethernet frames is shown in Table 19. An eth\_frame\_buffer list will be globally defined for storing the Ethernet frames. Depending on the Ethernet type requested, the *get\_next\_frame()* function will search through the eth\_frame\_buffer list and return the ethernet frame in the input buffer passed in from its caller. For example, the *get\_eth\_ip\_frame()* will request an Ethernet frame of Ethernet-IP type from *get\_next\_frame()* function. It will pass in an input buffer to *get\_next\_frame()* to store the Ethernet-IP frame. Whether or not the operation is able to retrieve the required Ethernet frame will be determined by the result code returned by *get\_next\_frame()* function.

eth_frame_buffer		
Type	Name	Description
struct eth_frame_buffer *	next	Pointer to next eth_frame_buffer of a linked list
struct pbuf*	p	Packet buffer
unsigned short	eth_type	Ethernet type
unsigned int	len	Length of frame
char [4]	netif_name	Name of netif eg. 'eth1'

Table 19. Data structure for Ethernet frame buffer

The *get\_next\_frame()* function is also designed to have the option of processing Ethernet-ARP type frame. If this option is chosen, the *get\_next\_frame()* function will call *ethernet\_input()* function, which then calls the *etharp\_arp\_input()* function to handle ARP type Ethernet frames. The *etharp\_arp\_input()* function will either send an ARP reply if the incoming packet is an ARP request from a remote host or updates the ARP cache if the incoming packet is an ARP reply from a remote host to the sender host.

Another change was made in relation to the processing of ARP packets during the sending of IP packets. In the previous design, the ARP request and ARP reply are not within the same functional flow. When the hardware address of the destination host is not in the ARP cache of the sender host, an ARP request is sent asking for the hardware address of the destination IP address. The sender host will receive an ARP reply in response to the ARP request sent. For the initial design, when the sender host sends an IP packet, it will send an ARP request if there is no hardware address based on the destination IP address in the host ARP cache. The flow for sending IP packets starts at the application layer and ends after the ARP request is sent and the IP packet with the proper Ethernet header information is sent to a queue. The queued IP packet can only be sent if the ARP cache is updated, which is done after receiving an ARP reply. To get the ARP reply, another function flow for receiving incoming packets has to be followed. The flow for receiving IP packet starts when *lpsk\_read\_next()* is called. In the context of the

ARP protocol, the flow ends when an ARP reply is received and has triggered a function call to update Ethernet header information with the destination hardware address before sending out the queued IP packet.

The function flow for sending IP packets does not include the processing of incoming ARP frames. Within this function flow, if there is an appropriate ARP entry in the ARP cache, the flow ends after the IP packet is sent. On the other hand, if there is no appropriate ARP entry in the ARP cache, an ARP request is sent, and the flow for sending IP packets ends here. This is not desired because the flow has to rely on another function flow for processing incoming ARP frames.

A change made in the final design addresses this issue and ensures that an ARP request is replied and an IP packet is sent within a single function flow. The modification is that after the sender host sends out an ARP request, the sending host will poll for any incoming ARP Ethernet frames using *get\_arp\_frame()* function. If it receives an ARP request, it will send out an ARP reply and continue to poll for the next ARP Ethernet frame until it receives its ARP reply. If it receives an ARP reply, it will update the ARP cache, fill in the necessary Ethernet header information, and send out the IP packet. By making sure that an ARP reply is received after an ARP request is sent, the amended design allows IP packets to be sent within the same function flow. The function flow for sending an IP packet is guaranteed to send an IP packet ultimately. Nevertheless, a mechanism must be in place to ensure that the sender host is not trapped polling for an ARP Ethernet frame if the ARP reply never reaches the sender host.

## C. SUMMARY

The requirements and design considerations for implementing an IP protocol stack in the LPSK environment were discussed in this chapter. The final design for the IP protocol stack ironed out the issue of sending an IP packet in one function flow and receiving an ARP reply in another function flow. IP protocol stack is implemented based on the final design and the implementation details are discussed in the next chapter.

## **IV. IMPLEMENTATION**

This chapter describes the implementation details of the IP protocol stack for the LPSK prototype. Section A describes the software and hardware environment used in the development. Section B describes the methodology used to implement the IP protocol stack.

### **A. DEVELOPMENT ENVIRONMENT**

The implementation and testing was carried out in a virtual machine environment. The virtualization software used was VMware Workstation version 7.1.0 [31] and was installed on a workstation running on Intel® Core™2 Quad 3GHz Processor, with 4.0 Gigabytes RAM and using the Windows 7 Professional [32] operating system. Two virtual machines, both running Fedora Core version 7 [33], were created. One of the virtual machines was used for development and the other virtual machine was used for deploying and testing the compiled binary. The IP Protocol stack was implemented as an application executing on the LPSK. The LPSK and the IP Protocol stack were compiled using the Open Watcom C compiler version 1.7 [34].

### **B. IMPLEMENTATION METHODOLOGY**

This section elaborates on the approach used during the implementation of the IP Protocol stack. Setting up the system to run Lightweight Internet Protocol (LWIP) will be discussed first. This is followed by a description on the miscellaneous system functions required to link the LWIP source code as well as the modifications made to the relevant LWIP functions as listed in Table 16 (See Chapter III). The implementation details for a ping application are discussed last.

#### **1. Configuring the LPSK for LWIP**

The source code for LWIP can be downloaded [24]. The version used in the development is version 1.4.0.rc1. To test whether LWIP can be compiled for execution in the LPSK context, the Makefile for compiling the LWIP files was modified. The LWIP

files essential for development of the IP stack are listed in Table 20. These files have to be included in the Makefile during compilation. Details of the changes to the Makefile can be found in Appendix A.

<b>File</b>	<b>File Location</b>
def.c	core/
init.c	core/
mem.c	core/
memp.c	core/
netif.c	core/
pbuf.c	core/
raw.c	core/
sys.c	core/
udp.c	core/
timers.c	core/
icmp.c	core/ipv4
ip.c	core/ipv4
inet.c	core/ipv4
ip_addr.c	core/ipv4
ip_frag.c	core/ipv4
inet_chksum.c	core/ipv4
etharp.c	netif/

Table 20. Essential LWIP files used

## 2. Miscellaneous System Functions

Several additional system functions are required by LWIP in order for its code to be linked without errors. These functions and their support categories are listed in Table 21. These functions are being implemented in another project [35]. For this development, they are all included in a separate source file (`clib.c`) so that the LWIP source code could reference these functions. Three header files: `stdio.h`, `string.h` and `stdlib.h`, were created to declare the function prototypes. These were placed in the same directory as the LPSK source files. The OpenWatcom compiler also has its own header files for `stdio.h`, `string.h` and `stdlib.h`. These OpenWatcom header

files have linkages to other functions and were not easily modified. It is also not a good idea to change the OpenWatcom header files as there may be other code that has dependencies on these header files. Instead, the `Makefile` file was modified to ensure that LWIP will bypass the three header files defined by the OpenWatcom compiler and use the new ones instead. These modifications are highlighted in Appendix A.

Function	Support Category
<code>memcpy</code>	Memory handling
<code>memset</code>	Memory handling
<code>strlen</code>	String handling
<code>fflush</code>	Standard input/output
<code>abort</code>	Process control handling
<code>printf</code>	Standard input/output

Table 21. Miscellaneous system functions required by LWIP

### 3. Modification of LWIP Functions

Most of the LWIP functions and data structures are used without modification. However, there are a few LWIP functions that need to be modified to be able to meet the final IP protocol stack design objectives shown in Figure 21 (See Chapter III). As mentioned earlier, the final design allows only functions in upper layers to call functions in the lower layers and not in the reverse direction. There are verification functions in `ip_input()` to check that the IP header is correctly formatted and handled. The basic checks are that the IP header size must not be larger than the packet size, the checksum of the IP header must be valid, and if the destination IP address is not for the receiving host, the IP packet is forwarded by `ip_forward()`. The `ip_input()` function, however, is designed to make up-calls to functions in the upper layer. For instance, when an IP packet is handled by `ip_input()`, which is in the network layer, a function call is made, depending on the type of IP protocol, to an upper layer such as the transport layer, the application layer or even an ICMP function (layer slightly above IP). A change made to `ip_input()`

disallows function calls to the upper layers. The basic verification functions in *ip\_input()* remain. The modified *ip\_input()* is renamed as *ip\_input\_verify()* to differentiate it from the original LWIP *ip\_input()* function.

Another modification is to the *etharp\_request()* function. The *etharp\_request()* function is called by the *etharp\_query()* function. As sending of an IP packet is only to be allowed in a single function flow, a slight amendment to the *etharp\_request()* function is to poll for a ARP reply before proceeding to send out the IP packet. If an ARP request comes in before an ARP reply, it will respond to the request by sending back an ARP reply. The original *etharp\_request()* will not poll for an ARP reply and therefore another function call flow is needed to receive ARP reply.

#### 4. Implementation of the Ping Application

Two LPSK partitions were set up and configured to test the IP protocol stack. The partitions are referred to as Net\_Prod (Network Producer) and Net\_Cons (Network Consumer). A *ping* application was created in PL3 for the test. If the IP protocol stack is successfully implemented, both partitions are able to ping each other. The existing LPSK is able to handle the sending and receiving of data between two devices. The devices are nonetheless virtual. The receiving and sending of data between the two devices is implemented using *lspk\_read\_next* and *lpsk\_write\_next* functions, respectively. Each partition will have its own device that is virtually connected to the device bound to the other partition.

In order to simulate two devices in a networked environment, they have to be set up to follow certain network configurations. The network topology of the two devices is illustrated in Figure 22. The device in the Net\_Prod partition is assigned with the IP address of 192.168.0.11. Its hardware address is 06:05:04:03:02:01 and device name is eth5. As for the device in Net\_Cons partition, it is given an IP address of 192.168.0.22. Its hardware address is 07:06:05:04:03:02 and device name is eth6. Both devices are assigned the same network mask of 255.255.255.0 and are connected to a gateway with an IP address of 192.168.0.1.

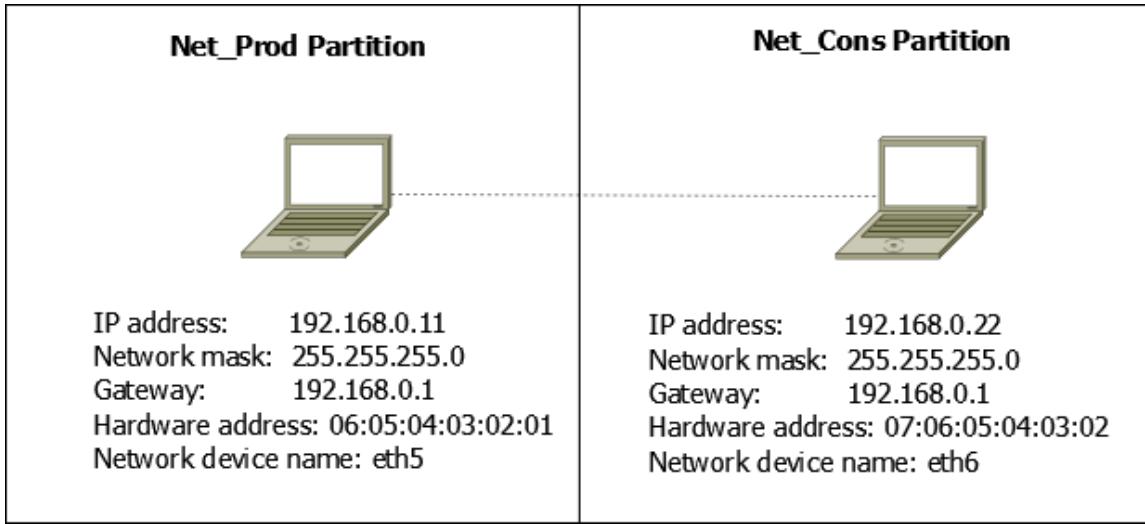


Figure 22. Network topology for Net\_Prod and Net\_Cons partitions

The final design in Figure 21 (See Chapter III) highlights the additional functions (marked in green) required in the implementation of the LPSK prototype IP protocol stack. In the final design, various functions are allocated to particular hardware privilege levels. However, the current implementation places all of these functions in PL3 together with the *ping* application. Although they occupy the same hardware privilege level, the functions are strictly layered and the functions are designed to only call functions belonging to a lower layer. In order to access data at the lower layer, a function relies on the mechanism in which buffer pointers are passed in as input parameters. Data is then copied into these buffers so that higher layer functions can access the data. The input parameters of these functions are listed in Table 22.

Function name	Input Parameters	
get_next_frame	void *const next_frame_buffer	Pointer to store received ethernet frame
	unsigned int *num_read	Pointer to store number of bytes read
	unsigned short eth_type	Ethernet type required
	void *const netif_name	Pointer to store network interface name of incoming packet
	unsigned int process_arp_flag	Flag to determine whether or not to process received ARP packets
get_eth_ip_frame	void *const buffer	Pointer to store received ethernet IP frame
	unsigned int *num_read	Pointer to store number of bytes read
	void *const netif_name	Pointer to store network interface name of incoming packet
get_ip_datagram	void *const buffer	Pointer to store received IP datagram
	unsigned int *num_read	Pointer to store number of bytes read
	void *const netif_name	Pointer to store network interface name of incoming packet
	unsigned char ip_protocol_type	The type of IP protocol required
get_icmp_packet	void *const buffer	Pointer to store received ICMP packet
	unsigned int *num_read	Pointer to store number of bytes read
	unsigned char icmp_type	ICMP type required
recv_ping	void *const buffer	Pointer to store received ping reply
	unsigned int *num_read	Pointer to store number of bytes read
send_icmp	ip_addr_t *src_ipaddr	Source IP address
	ip_addr_t *dest_ipaddr	Destination IP address
	unsigned char icmp_type	ICMP type to send
send_ping	ip_addr_t *src_ipaddr	Source IP address
	ip_addr_t *dest_ipaddr	Destination IP address

Table 22. Input parameters of additional functions

When a function in a lower layer is called by a function in an upper layer, a return value in the form of a result code is passed back. The return values are listed in Table 23 and are added to the existing `ip_stack.h` header file.

<b>Return Values</b>	<b>Description</b>
LPSK_ETHTYPE_DATA	A required type of Ethernet frame is in Ethernet frame buffer list
LPSK_ETHTYPE_ERROR	A required type of Ethernet frame is not in Ethernet frame buffer list
LPSK_ETH_IP_DATA	IP ethernet frame is in Ethernet frame buffer list
LPSK_ETH_IP_ERROR	IP ethernet frame is not in Ethernet frame buffer list
LPSK_IPTYPE_DATA	There is an IP datagram in IP datagram buffer list
LPSK_IPTYPE_ERROR	There is no IP datagram in IP datagram buffer list
LPSK_ICMPTYPE_DATA	The IP datagram matches the required ICMP type
LPSK_ICMPTYPE_ERROR	The IP datagram does not match the required ICMP type
LPSK_ETH_ARP_DATA	ARP ethernet frame is in Ethernet frame buffer list
LPSK_ETH_ARP_ERROR	ARP ethernet frame is not in Ethernet frame buffer list

Table 23. Return values between functions

## C. SUMMARY

*Ping* was implemented so that it can either send a ping request or receive a ping reply, or both in the same partition. To ensure that *ping* and the IP protocol stack work in the correct manner, testing is needed. The next chapter will discuss the test cases and test results used in the testing phase of this project.

**THIS PAGE INTENTIONALLY LEFT BLANK**

## V. TESTING

This chapter describes both the functional and acceptance test cases used for testing the functionality of the TCP/IP network stack. Section A describes the functional test cases while the acceptance test cases are presented in Section B. Section C describes the problems found during testing. Functional test cases are used to ensure that each function works as intended. Acceptance tests are used to verify that the overall system behaves appropriately. Exception tests are also included to ensure that the functions handle incorrect or out-of-range parameters properly. The results of the test cases will also be presented in this chapter. Test procedures and the detailed test results for each test case will appear in Appendix B.

### A. FUNCTIONAL TEST CASES

The original LWIP functions required for a ping application are tested first. Other tests include those for the functions that were created for this project or were modified from the original LWIP implementation. These functions are highlighted in green in Figure 21 (see Chapter III). The tests involve both functional tests and exception tests. Functional tests will be denoted with ‘F’ in the test type, and ‘E’ will represent an exception test. Tests for which the expected results match the actual results will be marked ‘Pass’. The tests will be marked ‘Fail’ otherwise. Two partitions are used for the tests. They are labeled Net\_Prod and Net\_Cons. Net\_Prod is used primarily to generate and send packets. Net\_Cons is used mainly to receive generated packets from Net\_Prod. In some cases, the tests require Net\_Cons to send packets over to Net\_Prod. The IP addresses assigned to Net\_Prod and Net\_Cons are 192.168.0.11 and 192.168.0.22, respectively.

#### 1. Original LWIP Functions

The function names in Figure 19 (see Chapter III) constitute the original LWIP functions. They are: *raw\_sendto()*, *ip\_route()*, *ip\_output\_if()*, *etharp\_output()*, *etharp\_query()*, *etharp\_request()*, *etharp\_send\_ip()*, *etharp\_arp\_input()*, *ethernet\_input()*, *ip\_input()*, *ip\_forward()*, *icmp\_input()*. The function *netif->linkoutput()*

is an interface to *lpsk\_write\_next()* and is assigned to the function *low\_level\_output()*. A ping application is used to test these functions. To ensure that the tests have provided adequate coverage, statements that print the function name when it is called are added to each function. A summary of the test and its results are listed in Table 24.

Test ID	Type	Test Description	Expected Results	Results (Pass/Fail)
FL1	F	Net_Prod sends a ping (ICMP Echo packet) to Net_Cons. Net_Cons sends a ping (ICMP Echo packet) to Net_Prod.	Both partitions receive ping replies from the other partition.	Pass

Table 24. Testing original LWIP functions

## 2. Newly Added and Modified LWIP Functions

The new functions as well as the modified LWIP functions are listed in Table 25. Each function is assigned a test group. Each test group, which represents a function, is used as a form of abbreviation for the function name. For each test group, exception testing and functional testing was conducted. The parameters used for each test and the corresponding test results are presented in Tables 26 through 36. The expected result of each test matches the observed result.

<b>Function</b>	<b>New / Modified</b>	<b>Test Group</b>
low_level_output	new	A
get_next_frame	new	B
get_eth_ip_frame	new	C
get_ip_datagram	new	D
get_icmp_packet	new	E
recv_ping	new	F
get_eth_arp_frame	new	G
ip_input_verify	modified	H
etharp_request	modified	I
send_icmp	new	J
send_ping	new	K

Table 25. Function test groupings

Function Test Group A: <i>netif-&gt;linkoutput()</i> (Equivalent to <i>low_level_output()</i> )				
Test ID	Type	Parameters	Expected Result	Result (Pass/Fail)
Fa1	E	*netif: NULL *pbuff: its payload contains Ethernet frame header	Error: Invalid parameter. netif is null. Data not sent.	Pass
Fa2	E	*netif: a valid netif to send from *pbuff: NULL	Error: Invalid parameter. pbuff is null. Data not sent.	Pass
Fa3	F	*netif: a valid netif to send from *pbuff: a valid pbuf with no payload (pbuff->length=0)	Function returns no error. The number of bytes written = 0	Pass
Fa4	F	*netif: a valid netif to send from *pbuff: its payload is Ethernet frame header	Data is sent without error. The number of bytes written = 14. An Ethernet frame with the same header information is received.	Pass

Table 26. Function test Group A – low\_level\_output()

Function Test Group B: <i>get_next_frame()</i>				
Test ID	Type	Parameters	Expected Result	Result (Pass/Fail)
Fb1	E	*next_frame_buffer: NULL *num_read: a valid int pointer eth_type: ETHTYPE_ARP *netif_name: a valid char pointer process_arp: FALSE ( or 0 )	Error: Invalid parameter. buffer pointer is null.	Pass
Fb2	E	*next_frame_buffer: a valid char pointer *num_read: NULL eth_type: ETHTYPE_ARP *netif_name: a valid char pointer process_arp: FALSE ( or 0 )	Error: Invalid parameter. num_read pointer is null.	Pass
Fb3	E	*next_frame_buffer: a valid char pointer *num_read: a valid int pointer eth_type: 999 (Invalid ETHTYPE) *netif_name: a valid char pointer process_arp: FALSE ( or 0 )	Error: Invalid parameter. Unknown Ethernet type requested.	Pass
Fb4	E	*next_frame_buffer: a valid char pointer *num_read: a valid int pointer eth_type: ETHTYPE_ARP *netif_name: NULL process_arp: FALSE ( or 0 )	Error: Invalid parameter. netif_name pointer is null.	Pass
Fb5	F	*next_frame_buffer: a valid char pointer	Precondition: An Ethernet ARP frame is sent	Pass

		<ul style="list-style-type: none"> <li>*num_read: a valid int pointer</li> <li>    eth_type: ETHTYPE_ARP</li> <li>*netif_name: a valid char pointer</li> <li>process_arp: FALSE ( or 0 )</li> </ul>	<p>without error.</p> <p>Expected: The ARP frame is added to the Ethernet frame buffer list. The ARP frame is retrieved but not processed.</p>	
Fb6	F	<ul style="list-style-type: none"> <li>*next_frame_buffer: a valid char pointer</li> <li>    *num_read: a valid int pointer</li> <li>        eth_type: ETHTYPE_ARP</li> <li>*netif_name: a valid char pointer</li> <li>process_arp: TRUE (or 1)</li> </ul>	<p>Precondition: An Ethernet ARP frame is sent without error.</p> <p>Expected: No ARP frame is added to the Ethernet frame buffer list. <i>ethernet_input()</i> is called to process the Ethernet ARP frame. An ARP frame is not returned in next_frame_buffer.</p>	Pass
Fb7	F	<ul style="list-style-type: none"> <li>*next_frame_buffer: a valid char pointer</li> <li>    *num_read: a valid int pointer</li> <li>        eth_type: ETHTYPE_IP</li> <li>*netif_name: a valid char pointer</li> <li>process_arp: FALSE ( or 0 )</li> </ul>	<p>Precondition: One Ethernet ARP frame and one Ethernet IP frame are sent without error.</p> <p>Expected: Both Ethernet ARP and IP frames are added to Ethernet frame buffer list. An Ethernet IP frame is retrieved. <i>ethernet_input()</i> is NOT called to process Ethernet ARP frame.</p>	Pass
Fb8	F	<ul style="list-style-type: none"> <li>*next_frame_buffer: a valid char pointer</li> <li>    *num_read: a valid int pointer</li> <li>        eth_type: ETHTYPE_IP</li> <li>*netif_name: a valid char pointer</li> <li>process_arp: TRUE (or 1)</li> </ul>	<p>Precondition: One Ethernet ARP frame and one Ethernet IP frame are sent without error.</p> <p>Expected: <i>ethernet_input()</i> is called to process the Ethernet ARP frame. Only an Ethernet IP frame is added to the Ethernet frame buffer list. The Ethernet IP frame is returned in next_frame_buffer.</p>	Pass

Table 27. Function test Group B – get\_next\_frame()

Function Test Group C: <i>get_eth_ip_frame()</i>				
Test ID	Type	Parameters	Expected Result	Result (Pass/Fail)
Fc1	E	*buffer: NULL *num_read: a valid int pointer *netif_name: a valid char pointer	Error: Invalid parameter. buffer pointer is null.	Pass
Fc2	E	*buffer: a valid char pointer *num_read: NULL *netif_name: a valid char pointer	Error: Invalid parameter. num_read pointer is null.	Pass
Fc3	E	*buffer: a valid char pointer *num_read: a valid int pointer *netif_name: NULL	Error: Invalid parameter. netif_name pointer is null.	Pass
Fc4	F	*buffer: a valid char pointer *num_read: a valid int pointer *netif_name: a valid char pointer	Precondition: One Ethernet ARP frame and one Ethernet IP frame are sent without error.  Expected: <i>ethernet_input()</i> is called to process the Ethernet ARP frame. Only the Ethernet IP frame is added to the Ethernet frame buffer list. <i>ip_input_verify()</i> is called to process the IP frame. The IP header information of the returned buffer is displayed.	Pass

Table 28. Function test Group C – *get\_eth\_ip\_frame()*

Function Test Group D: <i>get_ip_datagram()</i>				
Test ID	Type	Parameters	Expected Result	Result (Pass/Fail)
Fd1	E	*buffer: NULL *num_read: a valid int pointer *netif_name: a valid char pointer ip_protocol_type : IP_PROTO_ICMP	Error: Invalid parameter. buffer pointer is null.	Pass
Fd2	E	*buffer: a valid char pointer *num_read: NULL *netif_name: a valid char pointer ip_protocol_type : IP_PROTO_ICMP	Error: Invalid parameter. num_read pointer is null.	Pass
Fd3	E	*buffer: a valid char pointer *num_read: a valid int pointer *netif_name: NULL ip_protocol_type : IP_PROTO_ICMP	Error: Invalid parameter. netif_name pointer is null.	Pass
Fd4	E	*buffer: a valid char pointer *num_read: a valid int pointer *netif_name: a valid char pointer ip_protocol_type 11 (Invalid IP Protocol : Type)	Error: Invalid parameter. Unknown IP Protocol type.	Pass
Fd5	F	*buffer: a valid char pointer *num_read: a valid int pointer	Precondition: One Ethernet ARP frame and one Ethernet IP frame are sent without error. The protocol type of the	Pass

		<pre>*netif_name: a valid char pointer ip_protocol_type IP_PROTO_ICMP :</pre>	<p>IP datagram is UDP.</p> <p>Expected: <i>ethernet_input()</i> is called to process the Ethernet ARP frame. Only the Ethernet IP frame is added to the Ethernet frame buffer list. <i>ip_input_verify()</i> is called to process the IP frame. The received Ethernet IP frame is of the UDP protocol type, but the protocol type requested is ICMP, hence there is no matching Ethernet IP frame of ICMP type.</p>	
Fd6	F	<pre>*buffer: a valid char pointer *num_read: a valid int pointer *netif_name: a valid char pointer ip_protocol_type IP_PROTO_UDP :</pre>	<p>Precondition: One Ethernet ARP frame and one Ethernet IP frame are sent without error. The protocol type of the IP datagram is UDP.</p> <p>Expected: <i>ethernet_input()</i> is called to process the Ethernet ARP frame. Only the Ethernet IP frame is added to the Ethernet frame buffer list. <i>ip_input_verify()</i> is called to process the IP frame. A matching Ethernet IP frame of UDP type is returned. The IP header information of the returned buffer is displayed.</p>	Pass

Table 29. Function test Group D – get\_ip\_datagram()

Function Test Group E: <i>get_icmp_packet()</i>				
Test ID	Type	Parameters	Expected Result	Result (Pass/Fail)
Fe1	E	*buffer: NULL *num_read: a valid int pointer icmp_type: ICMP_ECHO	Error: Invalid parameter. buffer pointer is null.	Pass
Fe2	E	*buffer: a valid char pointer *num_read: NULL icmp_type: ICMP_ECHO	Error: Invalid parameter. num_read pointer is null.	Pass
Fe3	E	*buffer: a valid char pointer *num_read: a valid int pointer icmp_type: ICMP_TE (Other ICMP type)	Error: Invalid parameter. Unknown ICMP type.	Pass
Fe4	F	*buffer: a valid char pointer *num_read: a valid int pointer icmp_type: ICMP_ECHO	Precondition: One Ethernet IP (ICMP Echo Type) frame is sent without error.  Expected: The Ethernet IP frame is added to the Ethernet frame buffer list. <i>ip_input_verify()</i> is called to process the IP frame. A matching Ethernet IP frame of ICMP type is returned. The IP header and ICMP header information of the returned buffer are displayed.	Pass
Fe5	F	*buffer: a valid char pointer *num_read: a valid int pointer	Precondition: One Ethernet IP (ICMP Echo Type) frame is sent without error. Sending host is not configured to respond to ARP	Pass

		icmp_type: ICMP_ER (ER: Echo Reply)	<p>packets.</p> <p>Expected: The Ethernet IP frame is added to the Ethernet frame buffer list. <i>ip_input_verify()</i> is called to process the IP frame. <i>icmp_input()</i> is called to handle the ICMP message. An unresolved ARP request return code is expected since the sender host is not set up to respond to ARP.</p>	
Fe6	F	*buffer: a valid char pointer *num_read: a valid int pointer icmp_type: ICMP_ER (ER: Echo Reply)	<p>Precondition: One Ethernet IP (ICMP Echo Reply Type) frame is sent without error. Identifier for ICMP Echo reply message different from destination host.</p> <p>Expected: The Ethernet IP frame is added to the Ethernet frame buffer list. <i>ip_input_verify()</i> is called to process the IP frame. An Ethernet IP frame of ICMP Echo Reply type is returned. A warning message that the identifier of the ICMP Echo reply is mismatched is displayed. The IP header and ICMP header information of the returned buffer are displayed.</p>	Pass
Fe7	F	*buffer: a valid char pointer *num_read: a valid int pointer icmp_type: ICMP_ER (ER: Echo Reply)	<p>Precondition: One Ethernet IP (ICMP Echo Reply Type) frame is sent without error. Identifier for ICMP Echo reply message is the same as destination host.</p> <p>Expected: The Ethernet IP frame is added to the Ethernet frame buffer list. <i>ip_input_verify()</i> is called to process the IP frame. An Ethernet IP frame of ICMP Echo Reply type is returned. A message that affirms that the identifier of ICMP Echo reply matches is displayed. The IP header and ICMP header information of the returned buffer are displayed.</p>	Pass

Table 30. Function test Group E – get\_icmp\_packet()

Function Test Group F: <i>recv_ping()</i>				
Test ID	Type	Parameters	Expected Result	Result (Pass/Fail)
Ff1	E	*buffer: NULL *num_read: a valid int pointer	Error: Invalid parameter. buffer pointer is null.	Pass
Ff2	E	*buffer: a valid char pointer *num_read: NULL	Error: Invalid parameter. num_read is null.	Pass
Ff3	F	*buffer: a valid char pointer *num_read: a valid int pointer	Precondition: One Ethernet IP (ICMP Echo Reply Type) frame is sent without error. The Identifier for the ICMP Echo reply message is the same as that of the destination host.  Expected: The Ethernet IP frame is added to Ethernet frame buffer list. <i>ip_input_verify()</i> is called to process IP frame. An Ethernet IP frame of ICMP Echo Reply type is returned. A message that affirms that the identifier of the ICMP Echo reply matches is displayed. The IP header and ICMP header information of the returned buffer are displayed.	Pass

Table 31. Function test Group F – *recv\_ping()*

Function Test Group G: <i>get_eth_arp_frame()</i>				
Test ID	Type	Parameters	Expected Result	Result (Pass/Fail)
Fg1	F	(none)	<p>Precondition: One Ethernet ARP frame and one Ethernet IP frame are sent without error.</p> <p>Expected: Both the Ethernet ARP and IP frames are added to the Ethernet frame buffer list. The Ethernet IP frame is retrieved.  <code>ethernet_input()</code> is not called in <code>get_next_frame()</code> but it is called in <code>get_eth_arp_frame()</code> to process Ethernet ARP frame.</p>	Pass

Table 32. Function test Group G – *get\_eth\_arp\_frame()*

Function Test Group H: <i>ip_input_verify()</i>				
Test ID	Type	Parameters	Expected Result	Result (Pass/Fail)
Fh1	E	*pbuff: NULL *netif: a valid netif pointer	Error: Invalid parameter. pbuf pointer is null.	Pass
Fh2	E	*pbuff: a valid pbuf pointer *netif: NULL	Error: Invalid parameter. netif pointer is null.	Pass
Fh3	E	*pbuff: a valid pbuf pointer *netif: a valid netif pointer	Precondition: One Ethernet IP frame with an IP header length greater than pbuf size is sent.  Expected: The IP header length does not fit in pbuf length. The IP packet is dropped.	Pass
Fh4	E	*pbuff: a valid pbuf pointer *netif: a valid netif pointer	Precondition: One Ethernet IP frame with an IP total length greater than pbuf size is sent.  Expected: The IP total length does not fit in pbuf length. The IP packet is dropped.	Pass
Fh5	E	*pbuff: a valid pbuf pointer *netif: a valid netif pointer	Precondition: One Ethernet IP frame of destination IP address 192.168.0.99 (other IP address in same network) is sent.  Expected: The IP packet will be forwarded using <i>ip_forward()</i> and since there is no device having such an IP address, there will not be any ARP reply from the destination host. Hence the packet is dropped.	Pass

Fh6	E	*pbuf: a valid pbuf pointer *netif: a valid netif pointer	Precondition: One Ethernet IP frame of destination IP address 192.168.2.111 (an IP address in a different network) is sent.  Expected: An IP packet will be forwarded using <i>ip_forward()</i> and since there is no network interface in the same network, there will be no forwarding route for the destination IP address. The packet is dropped.	Pass
Fh7	E	*pbuf: a valid pbuf pointer *netif: a valid netif pointer	Precondition: One Ethernet IP frame with invalid IP header checksum is sent.  Expected: The checksum failed. The IP packet is dropped.	Pass
Fh8	F	*pbuf: a valid pbuf pointer *netif: a valid netif pointer	Precondition: One Ethernet IP frame with IP header length and IP total length not greater than pbuf size, that has a valid checksum, and is for destination host 192.168.0.22 is sent.  Expected: IP packet is checked successfully by <i>ip_input_verify()</i> .	Pass

Table 33. Function test Group H – *ip\_input\_verify()*

Function Test Group I: <i>etharp_request()</i>				
Test ID	Type	Parameters	Expected Result	Result (Pass/Fail)
Fi1	E	*netif: NULL *ipaddr: a valid ipaddr pointer	Expected: Error: Invalid parameter. Netif pointer is null.	Pass
Fi2	E	*netif: a valid netif pointer *ipaddr: NULL	Expected: Error: Invalid parameter. IP address is null.	Pass
Fi3	F	*netif: a valid netif pointer *ipaddr: a valid ipaddr pointer (ipaddr 192.168.0.22)	<p>Precondition: <i>etharp_request()</i> is called from the Net_Prod partition. <i>get_eth_arp_frame()</i> is called from the Net_Cons partition to process incoming ARP frames.</p> <p>Expected: The Net_Prod partition sends an ARP request and waits for its ARP reply. The ARP cache table in the Net_Prod partition is updated when it receives an ARP reply. The Net_Cons partition receives an ARP request from Net_Prod, adds an entry to its ARP cache table and sends an ARP reply back to Net_Prod.</p>	Pass

Table 34. Function test Group I – *etharp\_request()*

Function Test Group J: <i>send_icmp()</i>				
Test ID	Type	Parameters	Expected Result	Result (Pass/Fail)
Fj1	E	src_ipaddr: 192.168.0.11 dest_ipaddr: 192.168.0.22 icmp_type: ICMP_TE (Other ICMP type)	Precondition: The Net_Cons partition is set up to listen for an ICMP Echo message.  Expected: Error: Invalid parameter: This ICMP type is not supported. The only current support is for ICMP Echo messages.	Pass
Fj2	E	src_ipaddr: 192.168.0.11 dest_ipaddr: NULL icmp_type: ICMP_ECHO	Precondition: The Net_Cons partition is set up to listen for an ICMP Echo message.  Expected: The Destination IP address is 0.0.0.0. There is no network interface that belongs to same network as the Destination IP address 0.0.0.0, hence there is no route to the Destination IP address.	Pass
Fj3	E	src_ipaddr: 192.168.0.11 dest_ipaddr: 192.168.0.22 icmp_type: ICMP_ER (ER: Echo Reply)	Precondition: The Net_Cons partition is set up to listen for an ICMP Echo message.  Expected: Error: Invalid parameter: This ICMP type is not supported. The only current support is for ICMP Echo messages.	Pass
Fj4	F	src_ipaddr: 0.0.0.0 dest_ipaddr: 192.168.0.22	Precondition: The Net_Cons partition is set up to listen for an ICMP Echo message.	Pass

		icmp_type: ICMP_ECHO	Expected: Source IP address is 0.0.0.0. <i>raw_sendto()</i> is called to send ICMP Echo to the destination IP address 192.168.0.22. <i>raw_sendto()</i> checks for a network interface from which to send the packet out. The IP address of netif is used as the source IP address instead of 0.0.0.0	
Fj5	F	src_ipaddr: 192.168.0.11 dest_ipaddr: 192.168.0.22 icmp_type: ICMP_ECHO	Precondition: The Net_Cons partition is set up to listen for an ICMP Echo message.  Expected: An ICMP Echo Type message is sent from the IP address 192.168.0.11 to 192.168.0.22. An ICMP Echo Reply is sent back from 192.168.0.22. An ARP Reply is first returned to the host at 192.168.0.11 before the IP datagram is processed by the host at 192.168.0.22	Pass

Table 35. Function test Group J – send\_icmp()

Function Test Group K: <i>send_ping()</i>				
Test ID	Type	Parameters	Expected Result	Result (Pass/Fail)
Fk1	F	dest_ipaddr: 192.168.0.22	<p>Precondition: The Net_Cons partition is set up to listen for an ICMP Echo message.</p> <p>Expected: An ICMP Echo Type message is sent from IP address 192.168.0.11 to 192.168.0.22. An ICMP Echo Reply is sent back from 192.168.0.22. An ARP Reply is first returned to host at 192.168.0.11 before an IP datagram is processed by the host at 192.168.0.22</p>	Pass
Fk2	F	dest_ipaddr: 192.168.0.99 (other address in same network)	<p>Precondition: The Net_Cons partition is set up to listen for an ICMP Echo message.</p> <p>Expected: There is no device on the same network with IP address 192.168.0.99, hence the ARP request will time out and the packet is dropped.</p>	Pass

Fk3	F	192.168.0.11 dest_ipaddr: (its own IP address)	<p>Precondition: netif in the Net_Prod partition is configured to be used as a loopback device (minor_device[2]) to send and receive. Net_Prod will send and receive ping from same partition.</p> <p>Expected: Host 192.168.0.11 can send and receive ping (Echo and Echo Reply) to and from itself.</p>	Pass
Fk4	F	192.168.2.111 dest_ipaddr: (IP address in different network)	<p>Precondition: The Net_Cons partition is set up to listen for ICMP Echo message.</p> <p>Expected: There is no network interface belonging to the same network as 192.168.2.111, hence the packet is dropped.</p>	Pass

Table 36. Function test Group K – send\_ping()

## B. ACCEPTANCE TESTS

A ping application was developed for testing the overall network stack implemented in the LPSK. Two partitions were used for the ping application. The two partitions were named: Net\_Prod and Net\_Cons. Net\_Prod was given an IP address of 192.168.0.11 while Net\_Cons was assigned an IP address of 192.168.0.22. Depending on the objective of the acceptance test, each partition was configured accordingly.

### 1. Acceptance Test A1 and Results

The purpose of test A1 is to verify that when a partition (source host) sends an ICMP Echo request to another partition (remote host), it will receive an ICMP Echo Reply from the remote host. This test also ensures that the remote host will process incoming IP packets. If the remote host receives an ICMP Echo request, it will send an ICMP Echo reply back to the host requesting it. One partition (Net\_Prod) is set up to generate ping packets and to wait for ping replies. The other partition (Net\_Cons) serves as an active host that waits for incoming ping packets. Net\_Prod will ping Net\_Cons and wait for a reply from Net\_Cons. Upon receiving a Ping (ICMP Echo) packet, Net\_Cons will reply with an ICMP Echo Reply packet. The result of the test is shown in Table 37.

Test ID	Type	Test Description	Expected Results	Result (Pass/Fail)
A1	F	Net_Prod sends a ping (ICMP Echo packet) to Net_Cons.	Net_Prod sends a ping and receives a ping reply from Net_Cons.	Pass

Table 37. Acceptance test A1: Sending ICMP Echo from one host

### 2. Acceptance Test A2 and Results

The objective of test A2 is to verify that both partitions can send, receive and process ICMP packets. Similar to Net\_Prod in test A1, Net\_Cons is configured to send

ICMP Echo to Net\_Prod and to wait for ICMP Echo Reply from Net\_Prod. Net\_Cons is expected to receive an ICMP Echo from Net\_Prod and send an ICMP Echo Reply back to Net\_Prod.

In the other partition, when Net\_Prod receives an ICMP Echo packet, it will reply with ICMP Echo Reply to sender host. Net\_Prod is expected to send a ping and receive its ping reply from Net\_Cons. The description of the test and its results are summarized in Table 38.

. Test ID	Type	Test Description	Expected Results	Result (Pass/Fail)
A2	F	<p>(a) Net_Prod sends a ping (ICMP Echo packet) to Net_Cons and waits for ping reply.</p> <p>(b) Net_Cons sends a ping (ICMP Echo packet) to Net_Prod and waits for ping reply</p>	<p>Precondition: Both actions (a) and (b) runs simultaneously.</p> <p>Expected:</p> <p>(a) Net_Prod sends a ping and receives a ping reply from Net_Cons. Net_Prod receives a ping from Net_Cons and sends a ping reply back to Net_Cons.</p> <p>(b) Net_Cons sends a ping and receives a ping reply from Net_Prod. Net_Cons receives a ping from Net_Prod and sends a ping reply back to Net_Prod.</p>	Pass

Table 38. Acceptance test A2: Sending ICMP Echo between hosts

### 3. Acceptance Test A3 and Results

The purpose of Test A3 is to verify that there must be a device present that can respond when it is queried by another device. For this test, Net\_Prod sends a ping packet

to Net\_Cons. Net\_Cons, however, is not listening or responding to incoming packets. The expected result of this test is that Net\_Prod will not get a ping reply back from Net\_Cons. The result of this test is shown in Table 39.

<b>Test ID</b>	<b>Type</b>	<b>Test Description</b>	<b>Expected Results</b>	<b>Result (Pass/Fail)</b>
A3	E	Net_Prod sends a ping (ICMP Echo packet) to Net_Cons.	Precondition: Net_Cons is not listening or responding to ping packets.  Expected: The IP packet is dropped since Net_Prod cannot receive its ARP reply from Net_Cons. Therefore, there is no ping reply received from Net_Cons.	Pass

Table 39. Acceptance test A3: Sending ICMP Echo to non-responding host

## C. PROBLEMS ENCOUNTERED

There were a few problems discovered during the acceptance testing. The problems are discussed in this section.

### 1. Memory Allocation

One of the problems encountered during testing was that when the *ping* application sent too many ping packets, the application crashed. A closer look at the problem suggests that the memory allocated to the pbufs in the application was not properly de-allocated hence causing the application to exhaust its allocated memory. An attempt was made to verify this claim by doubling the allocated heap memory used by the application. In LWIP, the size of the heap memory used by an application can be changed by modifying the value for the `MEM_SIZE` in the header file `lwipopts.h`. Before the memory size was increased, the number of ping packets the application could handle was about 25 (using Test A1). After doubling the allocated heap memory size, the number of ping packets increased to 50 before the application crashed. Apparently, the memory used

up by the application was not released after each ping is sent. When the allocated heap memory was doubled again, another problem related to LWIP heap memory allocation was found: the *ping* application crashed during boot up. .

## **2. LWIP Heap Memory**

The ping application crashed during boot up when the LWIP heap memory is set to a value larger than 64000 bytes. Preliminary investigation of this problem suggests that it may be related to the memory functions in `mem.c`, which is used by LWIP for doing memory operations. The problem may be caused by pointer assignments and pointer arithmetic used in LWIP. The LPSK uses far pointers for its memory model while LWIP may use near pointers. A far pointer has a segment selector and an offset value, which allows the LPSK to use a memory region in a different memory segment. A near pointer, on the other hand, only has an offset value, which references the current memory segment. A problem may occur if the offset value of the far pointer refers to a different memory segment but is interpreted as a near pointer, i.e., to be the offset value within the current memory segment. The offset value may be larger than the size of current memory segment and cause the application to crash. More study, however, is needed to understand the memory allocation problem completely. An approach analyzing the problem is to determine the offset value used during pointer arithmetic by LWIP, and whether or not the result falls within the range of current LPSK heap segment.

## **D. SUMMARY**

The functional tests and acceptance tests and their results were presented in this chapter. All the tests passed, i.e. the expected results match the actual results. The test procedures and actual test results for all the tests can be found in Appendix B. Two problems relating to memory were discovered during the acceptance testing. They were most likely caused by improper de-allocation of memory and misinterpretation of offset value between near and far pointers.

## **VI. CONCLUSION AND FUTURE WORK**

Transient access to sensitive information during critical situations allows soldiers to have a tactical advantage in a battle. Accessing sensitive information using a tactical device at critical moments enables soldiers to be more aware of their situational surroundings and allows them to make well-informed decisions. A tactical device, that contains sensitive information, not only must have the capability to protect the information, it is also desirable to provide transient access to information to soldiers at critical junctures. The use of a separation kernel with appropriate protected services and multilevel secure services can support such a paradigm. The objective of this work has been to support such capabilities with a networking functionality. This chapter concludes this thesis and provides some suggestions for future work.

### **A. CONCLUSION**

This project is focused on implementing an IP protocol stack for the LPSK. An evaluation of open source implementations of network protocol stacks was carried out during the design phase. A suitable open source implementation, LWIP, was eventually chosen, but some modifications were needed to fit it into the overall LPSK architecture. LPSK architecture is organized so that functions are decomposed into modules and layered in such a way that functions in the upper layers can only make calls to functions in the lower layers but not the other way round, i.e. there are no circular dependencies among the functions. The original LWIP architecture, however, allows lower layers to make function calls to the upper layers. Such inconsistency is resolved in the final design of the IP protocol stack. The IP protocol stack, which uses some of LWIP functions, was successfully implemented and integrated with the existing LPSK.

Functional and acceptance tests were conducted. Each of the functions, which were either newly created or modified from LWIP original functions, was tested for its correctness and its ability to handle exceptions during the functional test. For the acceptance tests, a ping application was developed as a system prototype for a networking demonstration using the LPSK IP protocol stack. The system prototype runs

and executes successfully. To conclude, this project affirms the hypothesis that an IP protocol stack can be built and used to simulate network traffic in a MLS system prototype, and that the IP protocol stack was able to be modularized and layered to fit the LPSK architecture.

With the implemented IP protocol stack, there is now a higher chance of providing networking functionality in MLS tactical hand held devices. Full networking capability can only be achieved if the network stack provides support for the whole TCP/IP protocol suite, and if driver for networking hardware such as network interface controller is implemented. The IP protocol stack can only simulate ping within the MLS prototype, but with future enhancements, it is possible to provide networking functionality in MLS device.

## **B. FUTURE WORK**

The following areas can be considered for future enhancements of the IP protocol stack.

### **1. Separation of Privilege Levels**

The current implementation of the IP protocol stack is not separated into hardware privilege levels although its functions are modularized and layered. A study of how each layer of the IP protocol stack should be separated to different hardware privilege levels can be performed. The effects of assigning a particular protocol stack layer to different privilege levels can also be studied.

### **2. Transport Layer Services Support**

The original LWIP includes functions to support operations of the transport layer protocols, in particular TCP and UDP, but they were not implemented in the LPSK IP protocol stack. To extend the capability of the protocol stack in the LPSK, support for TCP and UDP can be considered. With TCP and UDP support available, applications that require TCP or UDP, such as HyperText Transfer Protocol (HTTP), File Transfer Protocol (FTP), and Dynamic Host Configuration Protocol (DHCP), can then be supported.

### **3. Network Layer Services Extension**

The IP protocol stack only supports Internet Protocol version 4. As more systems are gearing toward the next generation Internet Protocol, which is IPv6, adding support for IPv6 to the IP protocol stack can be considered. The IP protocol stack currently supports ICMP version 4 (ICMPv4). ICMPv6 should also be considered along with IPv6 as IPv6 cannot support ICMPv4. In addition, support for IPSec can also be considered. IPSec is a mechanism that “provides security to IP and upper-layer protocols” [36]. In addition, the current ICMP types supported are Echo and Echo Reply. Other ICMP types such as Timestamp Request and Reply, and Address Mask Request and Reply can be considered.

**THIS PAGE INTENTIONALLY LEFT BLANK**

## APPENDIX A: MAKEFILE CONFIGURATION

This appendix lists the changes added to the Makefile to compile the LWIP files used in this project as well as for compiling the code for the IP stack and the *ping* application. The changes added to Makefile are as follows:

```
N_PROD_EXE = net_prod
N_PROD_OBJS = net_prod.o tsm_stack.o tsm_io.o
$(LPSK_IP_STACK_OBJS)
FN_PROD_OBJS = FILE net_prod.o,tsm_stack.o,tsm_io.o,
$(LPSK_IP_STACK_FOBJS)

N_CONS_EXE = net_cons
N_CONS_OBJS = net_cons.o tsm_stack.o tsm_io.o
$(LPSK_IP_STACK_OBJS)
FN_CONS_OBJS = FILE net_cons.o,tsm_stack.o,tsm_io.o,
$(LPSK_IP_STACK_FOBJS)

LWIP_CORE_OBJS = def.o init.o mem.o memp.o netif.o pbuf.o raw.o
sys.o udp.o timers.o

LWIP_CORE_FOBJS = def.o,init.o,mem.o,memp.o,netif.o,pbuf.o,
raw.o,sys.o,udp.o,timers.o

LWIP_IP4_OBJS = icmp.o ip.o inet.o ip_addr.o ip_frag.o
inet_chksum.o

LWIP_IP4_FOBJS =
icmp.o,ip.o,inet.o,ip_addr.o,ip_frag.o,inet_chksum.o

LWIP_NETIF_OBJS = etharp.o

LWIP_NETIF_FOBJS = etharp.o

LPSK_IP_STACK_OBJS = ip_stack.o clib.o $(LWIP_CORE_OBJS)
$(LWIP_IP4_OBJS) $(LWIP_NETIF_OBJS)

LPSK_IP_STACK_FOBJS = ip_stack.o,clib.o,$(LWIP_CORE_FOBJS),
$(LWIP_IP4_FOBJS),$(LWIP_NETIF_FOBJS)

#changes to use new string.h, stdlib.h, stdio.h
INC1 = -
i.:./include:./include/lwip:./arch:./include/ipv4:./include/snmp:
./include/netif:$(_WATCOM)/lh

ip_stack.o:      wcc386 ip_stack.c $(INC1) $(CC_OPTS)
```

```

clib.o:      wcc386 clib.c $(INC1) $(CC_OPTS)
etharp.o:    wcc386 netif/etharp.c $(INC1) $(CC_OPTS)
icmp.o:     wcc386 core/ipv4/icmp.c $(INC1) $(CC_OPTS)
ip.o:        wcc386 core/ipv4/ip.c $(INC1) $(CC_OPTS)
inet.o:      wcc386 core/ipv4/inet.c $(INC1) $(CC_OPTS)
ip_addr.o:   wcc386 core/ipv4/ip_addr.c $(INC1) $(CC_OPTS)
ip_frag.o:   wcc386 core/ipv4/ip_frag.c $(INC1) $(CC_OPTS)
inet_chksum.o: wcc386 core/ipv4/inet_chksum.c $(INC1) $(CC_OPTS)
def.o:       wcc386 core/def.c $(INC1) $(CC_OPTS)
netif.o:     wcc386 core/netif.c $(INC1) $(CC_OPTS)
pbuf.o:      wcc386 core/pbuf.c $(INC1) $(CC_OPTS)
sys.o:       wcc386 core/sys.c $(INC1) $(CC_OPTS)
mem.o:       wcc386 core/mem.c $(INC1) $(CC_OPTS)
raw.o:       wcc386 core/raw.c $(INC1) $(CC_OPTS)
timers.o:   wcc386 core/timers.c $(INC1) $(CC_OPTS)
init.o:      wcc386 core/init.c $(INC1) $(CC_OPTS)
memp.o:     wcc386 core/memp.c $(INC1) $(CC_OPTS)
udp.o:      wcc386 core/udp.c $(INC1) $(CC_OPTS)

net_prod :  $(N_PROD_OBJS)
               wlink name $(N_PROD_EXE) debug all system lxkernel \
               option noextension option internalrelocs \
               option verbose option start=_net_prod_main option map\
               option maxerrors=250 option symfile \
               $(FN_PROD_OBJS) LIBFILE $(EXE).lib, $(PL2_EXE).lib
               chmod 440 $@

net_cons :  $(N_CONS_OBJS)
               wlink name $(N_CONS_EXE) debug all system lxkernel \
               option noextension option internalrelocs \
               option verbose option start=_net_cons_main option map\
               option maxerrors=250 option symfile \
               $(FN_CONS_OBJS) LIBFILE $(EXE).lib, $(PL2_EXE).lib
               chmod 440 $@

all:        $(PROCESS_CALLGATES)      $(PROCESS_KEYMAP)      $(EXE)
$(PL1_EXE)    $(PL2_EXE)      $(GATES)      $(PL1_GATES)      $(PL2_GATES)
$(TSM_EXE)    $(TPA_EXE)      $(BOX_EXE)      $(CLOCK_EXE)      $(TSM_SC_EXE)
$(TSM2_EXE)   $(KEYMAP_OUTPUT_FILE)  $(N_PROD_EXE)  $(N_CONS_EXE)

```

## **APPENDIX B: TEST PROCEDURES**

Appendix B describes the procedures for the Testing phase described in Chapter V. The expected results of each test for each partition are also presented.

### **A. Test Procedures for Original LWIP Functions**

1. Go to '`<current-working-dir>/tcx/trunk/kernel/2tests/original-lwip`' directory. `<current-working-dir>` is where you installed the LPSK source code.
2. Build the source code by executing the command '`make clean all`'.
3. Upload the compiled binary to the test machine.
4. Start the test machine. At the Login screen, enter the correct username and password.
5. At the main menu, press 'F' for "change partition Focus."
6. Key in '1' to select "Network Producer" partition.
7. Observe results in "Network Producer" partition.
8. To change partition, press the Secure Attention Key combination: `Alt+Esc`
9. Key in '2' to select "Network Consumer" partition.
10. Observe results in "Network Consumer" partition.

### **B. Test Procedures for Newly Added and Modified LWIP Functions**

1. Go to '`<current-working-dir>/tcx/trunk/kernel`' directory. `<current-working-dir>` is where you installed the LPSK source code.
2. Run the script '`loadtest.sh`' to copy files from the sub-directory '`2tests`'.  
For example, if the test ID is Fa1, enter the command '`./loadtest.sh Fa1`'.
  - a. Original `net_prod.c` file is renamed as `net_prod.c.backup`.
  - b. Original `net_cons.c` file is renamed as `net_cons.c.backup`.

3. File ‘net\_prod.c.Fa1’ from ‘2tests’ sub-directory will be copied over to the ‘<current-working-dir>/tcx/trunk/kernel’ directory as ‘net\_prod.c’.
4. File ‘net\_cons.c.Fa1’ from ‘2tests’ sub-directory will be copied over to the ‘<current-working-dir>/tcx/trunk/kernel’ directory as ‘net\_cons.c’.
5. Build the source code by running the script ‘./build’.
6. Upload the compiled binary to the test machine.
7. Start the test machine. At the Login screen, enter the correct username and password.
8. At the main menu, press ‘F’ for “change partition Focus.”
9. Key in ‘1’ to select “Network Producer” partition.
10. Observe results in “Network Producer” partition.
11. To change partition, press the Secure Attention Key combination: Alt+Esc
12. Key in ‘2’ to select “Network Consumer” partition.
13. Observe results in “Network Consumer” partition.
14. To restore the original net\_prod.c and net\_cons.c files,
  - a. Rename net\_prod.c.backup to net\_prod.c
  - b. Rename net\_cons.c.backup to net\_cons.c
15. To change to another test, turn off the test machine, start the development machine. Repeat the steps 1 to 14.

## C. Functional Test Expected Results

This section lists the expected results of Net\_Prod and Net\_Cons partitions for each functional test.

Function Test: Original LWIP Functions		
Test ID	Expected Result (Net_Prod)	Expected Result (Net_Cons)
FL1	etharp_request() low_level_output() raw_sendto()	etharp_request() raw_sendto() ip_route()

	ip_route() ip_output_if() etharp_output() etharp_query() etharp_request() low_level_output() [192.168.0.11] Ping 192.168.0.22... etharp_send_ip() low_level_output() ethernet_input() etharp_arp_input() low_level_output() ethernet_input() ip_input() icmp_input() Version: 4 Protocol: 1 Source IP: 192.168.0.22 Dest IP: 192.168.0.11 ICMP Type: 8, code: 0, id:16822 ip_output_if() etharp_output() etharp_send_ip() low_level_output() ethernet_input() etharp_arp_input() ethernet_input() ip_input() icmp_input() Version: 4 Protocol: 1 Source IP: 192.168.0.22 Dest IP: 192.168.0.11 ICMP Type: 0, code: 0, id:16811	ip_output_if() etharp_output() etharp_query() etharp_request() [192.168.0.22] Ping 192.168.0.11... ethernet_input() etharp_arp_input() etharp_send_ip() ethernet_input() etharp_arp_input() ethernet_input() ip_input() icmp_input() Version: 4 Protocol: 1 Source IP: 192.168.0.11 Dest IP: 192.168.0.22 ICMP Type: 8, code: 0, id:16811 ip_output_if() etharp_output() etharp_send_ip() ethernet_input() etharp_arp_input() ethernet_input() ip_input() icmp_input() Version: 4 Protocol: 1 Source IP: 192.168.0.11 Dest IP: 192.168.0.22 ICMP Type: 0, code: 0, id:16822
--	---	---

Function Test Group A: <i>low_level_output()</i>		
Test ID	Expected Result (Net_Prod)	Expected Result (Net_Cons)
Fa1	netif name: eth5 [Eth src] 11:22:33:44:55:11 [Eth dest] 1:2:3:4:5:1 Error: Invalid parameter. netif is null.	num of bytes read = 0
Fa2	netif name: eth5 [Eth src] 11:22:33:44:55:11 [Eth dest] 1:2:3:4:5:1	num of bytes read = 0

	Error: Invalid parameter. pbuf is null.	
Fa3	netif name: eth5 Length of pbuf = 0 Total length of pbuf + chain = 0 no of bytes written = 0 Ethernet ARP frame sent	num of bytes read = 0
Fa4	netif name: eth5 [Eth src] 11:22:33:44:55:11 [Eth dest] 1:2:3:4:5:1 [Eth type] ARP Length of pbuf = 14 Total length of pbuf + chain = 14 no of bytes written = 14 Ethernet ARP frame sent	skip ethernet_input() Eth [ARP] frame added to list num of bytes read = 14 LPSK_ETHTYPE_DATA netif name: eth6 [Eth src] 11:22:33:44:55:11 [Eth dest] 1:2:3:4:5:1 [Eth type] ARP

Function Test Group B: <i>get_next_frame()</i>		
Test ID	Expected Result (Net_Prod)	Expected Result (Net_Cons)
Fb1	netif name: eth5 [Eth src] 11:22:33:44:55:11 [Eth dest] 1:2:3:4:5:1 [Eth type] ARP Length of pbuf = 14 Total length of pbuf + chain = 14 no of bytes written = 14	Error: Invalid parameter. buffer pointer is null.
Fb2	netif name: eth5 [Eth src] 11:22:33:44:55:11 [Eth dest] 1:2:3:4:5:1 [Eth type] ARP Length of pbuf = 14 Total length of pbuf + chain = 14 no of bytes written = 14	Error: Invalid parameter. num_read pointer is null.
Fb3	netif name: eth5 [Eth src] 11:22:33:44:55:11 [Eth dest] 1:2:3:4:5:1 [Eth type] ARP Length of pbuf = 14 Total length of pbuf + chain = 14 no of bytes written = 14	Error: Invalid parameter. Unknown Ethernet type requested.
Fb4	netif name: eth5 [Eth src] 11:22:33:44:55:11 [Eth dest] 1:2:3:4:5:1 [Eth type] ARP	Error: Invalid parameter. netif_name pointer is null.

	Length of pbuf = 14 Total length of pbuf + chain = 14 no of bytes written = 14	
Fb5	netif name: eth5 [Eth src] 11:22:33:44:55:11 [Eth dest] 1:2:3:4:5:1 [Eth type] ARP Length of pbuf = 14 Total length of pbuf + chain = 14 no of bytes written = 14	skip ethernet_input() Eth [ARP] frame added to list. LPSK_ETHTYPE_DATA num of bytes read = 14 netif name: eth6 [Eth src] 11:22:33:44:55:11 [Eth dest] 1:2:3:4:5:1 [Eth type] ARP
Fb6	netif name: eth5 [Eth src] 11:22:33:44:55:11 [Eth dest] 1:2:3:4:5:1 [Eth type] ARP Length of pbuf = 14 Total length of pbuf + chain = 14 no of bytes written = 14	begin ethernet_input() ethernet_input() returns ERR_OK end ethernet_input() LPSK_TYPE_ERROR
Fb7	netif name: eth5 [Eth src] 99:88:77:66:55:44 [Eth dest] 9:8:7:6:5:4 [Eth type] IP Length of pbuf = 34 Total length of pbuf + chain = 34 IP packets sent.... Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22 Protocol: 17 (UDP) no of bytes written = 34 Ethernet IP frame sent  netif name: eth5 [Eth src] 11:22:33:44:55:11 [Eth dest] 1:2:3:4:5:1 [Eth type] ARP Length of pbuf = 14 Total length of pbuf + chain = 14 no of bytes written = 14 Ethernet ARP frame sent	Eth [IP] frame added to list skip ethernet_input() Eth [ARP] frame added to list. LPSK_ETHTYPE_DATA num of bytes read = 34 netif name: eth6 [Eth src] 99:88:77:66:55:44 [Eth dest] 9:8:7:6:5:4 [Eth type] IP IP packets received.... Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22 Protocol: 17 (UDP)
Fb8	netif name: eth5 [Eth src] 99:88:77:66:55:44 [Eth dest] 9:8:7:6:5:4 [Eth type] IP	Eth [IP] frame added to list begin ethernet_input() ethernet_input() returns ERR_OK end ethernet_input()

	<p>Length of pbuf = 34          Total length of pbuf + chain = 34          IP packets sent....          Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22          Protocol: 17 (UDP)          no of bytes written = 34          Ethernet IP frame sent</p> <p>netif name: eth5          [Eth src] 11:22:33:44:55:11          [Eth dest] 1:2:3:4:5:1          [Eth type] ARP          Length of pbuf = 14          Total length of pbuf + chain = 14          no of bytes written = 14          Ethernet ARP frame sent</p>	<p>LPSK_ETHTYPE_DATA          num of bytes read = 34          netif name: eth6          [Eth src] 99:88:77:66:55:44          [Eth dest] 9:8:7:6:5:4          [Eth type] IP          IP packets received....          Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22          Protocol: 17 (UDP)</p>
--	---	--

Function Test Group C: <i>get_ether_ip_frame()</i>		
Test ID	Expected Result (Net_Prod)	Expected Result (Net_Cons)
Fc1	<p>netif name: eth5          [Eth src] 99:88:77:66:55:44          [Eth dest] 9:8:7:6:5:4          [Eth type] IP          Length of pbuf = 34          Total length of pbuf + chain = 34          IP packets sent....          Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22          Protocol: 17 (UDP)          no of bytes written = 34          Ethernet IP frame sent</p> <p>netif name: eth5          [Eth src] 11:22:33:44:55:11          [Eth dest] 1:2:3:4:5:1          [Eth type] ARP          Length of pbuf = 14          Total length of pbuf + chain = 14          no of bytes written = 14          Ethernet ARP frame sent</p>	Error: Invalid parameter. buffer pointer is null.
Fc2	netif name: eth5	Error: Invalid parameter. num_read

	<p>[Eth src] 99:88:77:66:55:44  [Eth dest] 9:8:7:6:5:4  [Eth type] IP  Length of pbuf = 34  Total length of pbuf + chain = 34  IP packets sent....  Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22  Protocol: 17 (UDP)  no of bytes written = 34  Ethernet IP frame sent</p> <p>netif name: eth5  [Eth src] 11:22:33:44:55:11  [Eth dest] 1:2:3:4:5:1  [Eth type] ARP  Length of pbuf = 14  Total length of pbuf + chain = 14  no of bytes written = 14  Ethernet ARP frame sent</p>	pointer is null.
Fc3	<p>netif name: eth5  [Eth src] 99:88:77:66:55:44  [Eth dest] 9:8:7:6:5:4  [Eth type] IP  Length of pbuf = 34  Total length of pbuf + chain = 34  IP packets sent....  Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22  Protocol: 17 (UDP)  no of bytes written = 34  Ethernet IP frame sent</p> <p>netif name: eth5  [Eth src] 11:22:33:44:55:11  [Eth dest] 1:2:3:4:5:1  [Eth type] ARP  Length of pbuf = 14  Total length of pbuf + chain = 14  no of bytes written = 14  Ethernet ARP frame sent</p>	Error: Invalid parameter. netif_name pointer is null.
Fc4	netif name: eth5 [Eth src] 99:88:77:66:55:44	Eth [IP] frame added to list begin ethernet_input()

	<p>[Eth dest] 9:8:7:6:5:4  [Eth type] IP  Length of pbuf = 34  Total length of pbuf + chain = 34  IP packets sent....  Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22  Protocol: 17 (UDP)  no of bytes written = 34  Ethernet IP frame sent</p> <p>netif name: eth5  [Eth src] 11:22:33:44:55:11  [Eth dest] 1:2:3:4:5:1  [Eth type] ARP  Length of pbuf = 14  Total length of pbuf + chain = 14  no of bytes written = 14  Ethernet ARP frame sent</p>	<p>ethernet_input() returns ERR_OK  end ethernet_input()  begin ip_input_verify()  ip_input_verify() returns ERR_OK  end ip_input_verify()  LPSK_ETH_IP_DATA  num of bytes read = 34  netif name: eth6  IP packets received....  Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22  Protocol: 17 (UDP)</p>
--	--	--

Function Test Group D: <i>get_ip_datagram()</i>		
Test ID	Expected Result (Net_Prod)	Expected Result (Net_Cons)
Fd1	<p>netif name: eth5  [Eth src] 99:88:77:66:55:44  [Eth dest] 9:8:7:6:5:4  [Eth type] IP  Length of pbuf = 34  Total length of pbuf + chain = 34  IP packets sent....  Version: 4, Src IP: 192.168.0.11,  Dest IP: 192.168.0.22  Protocol: 17 (UDP)  no of bytes written = 34  Ethernet IP frame sent</p> <p>netif name: eth5  [Eth src] 11:22:33:44:55:11  [Eth dest] 1:2:3:4:5:1  [Eth type] ARP  Length of pbuf = 14  Total length of pbuf + chain = 14  no of bytes written = 14</p>	Error: Invalid parameter. buffer pointer is null.

	Ethernet ARP frame sent	
Fd2	<p>netif name: eth5  [Eth src] 99:88:77:66:55:44  [Eth dest] 9:8:7:6:5:4  [Eth type] IP  Length of pbuf = 34  Total length of pbuf + chain = 34  IP packets sent....  Version: 4, Src IP: 192.168.0.11,  Dest IP: 192.168.0.22  Protocol: 17 (UDP)  no of bytes written = 34  Ethernet IP frame sent</p> <p>netif name: eth5  [Eth src] 11:22:33:44:55:11  [Eth dest] 1:2:3:4:5:1  [Eth type] ARP  Length of pbuf = 14  Total length of pbuf + chain = 14  no of bytes written = 14  Ethernet ARP frame sent</p>	Error: Invalid parameter. num_read pointer is null.
Fd3	<p>netif name: eth5  [Eth src] 99:88:77:66:55:44  [Eth dest] 9:8:7:6:5:4  [Eth type] IP  Length of pbuf = 34  Total length of pbuf + chain = 34  IP packets sent....  Version: 4, Src IP: 192.168.0.11,  Dest IP: 192.168.0.22  Protocol: 17 (UDP)  no of bytes written = 34  Ethernet IP frame sent</p> <p>netif name: eth5  [Eth src] 11:22:33:44:55:11  [Eth dest] 1:2:3:4:5:1  [Eth type] ARP  Length of pbuf = 14  Total length of pbuf + chain = 14  no of bytes written = 14  Ethernet ARP frame sent</p>	Error: Invalid parameter. netif_name pointer is null.
Fd4	netif name: eth5 [Eth src] 99:88:77:66:55:44	Error: Invalid parameter. Unknown IP Protocol type.

	<p>[Eth dest] 9:8:7:6:5:4  [Eth type] IP  Length of pbuf = 34  Total length of pbuf + chain = 34  IP packets sent....  Version: 4, Src IP: 192.168.0.11,  Dest IP: 192.168.0.22  Protocol: 17 (UDP)  no of bytes written = 34  Ethernet IP frame sent</p> <p>netif name: eth5  [Eth src] 11:22:33:44:55:11  [Eth dest] 1:2:3:4:5:1  [Eth type] ARP  Length of pbuf = 14  Total length of pbuf + chain = 14  no of bytes written = 14  Ethernet ARP frame sent</p>	
Fd5	<p>netif name: eth5  [Eth src] 99:88:77:66:55:44  [Eth dest] 9:8:7:6:5:4  [Eth type] IP  Length of pbuf = 34  Total length of pbuf + chain = 34  IP packets sent....  Version: 4, Src IP: 192.168.0.11,  Dest IP: 192.168.0.22  Protocol: 17 (UDP)  no of bytes written = 34  Ethernet IP frame sent</p> <p>netif name: eth5  [Eth src] 11:22:33:44:55:11  [Eth dest] 1:2:3:4:5:1  [Eth type] ARP  Length of pbuf = 14  Total length of pbuf + chain = 14  no of bytes written = 14  Ethernet ARP frame sent</p>	<p>Eth [IP] frame added to list  begin ethernet_input()  ethernet_input() returns ERR_OK  end ethernet_input()  begin ip_input_verify()  ip_input_verify() returns ERR_OK  end ip_input_verify()  LPSK_IPTYPE_ERROR</p>
Fd6	<p>netif name: eth5  [Eth src] 99:88:77:66:55:44  [Eth dest] 9:8:7:6:5:4  [Eth type] IP  Length of pbuf = 34</p>	<p>Eth [IP] frame added to list  begin ethernet_input()  ethernet_input() returns ERR_OK  end ethernet_input()  begin ip_input_verify()</p>

	<p>Total length of pbuf + chain = 34      IP packets sent....      Version: 4, Src IP: 192.168.0.11,      Dest IP: 192.168.0.22      Protocol: 17 (UDP)      no of bytes written = 34      Ethernet IP frame sent</p> <p>netif name: eth5      [Eth src] 11:22:33:44:55:11      [Eth dest] 1:2:3:4:5:1      [Eth type] ARP      Length of pbuf = 14      Total length of pbuf + chain = 14      no of bytes written = 14      Ethernet ARP frame sent</p>	ip_input_verify() returns ERR_OK end ip_input_verify() LPSK_IPTYPE_DATA num of bytes read = 34 Netif name = eth6 IP packets received.... Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22 Protocol: 17 (UDP)
--	---	---

Function Test Group E: get_icmp_packet()		
Test ID	Expected Result (Net_Prod)	Expected Result (Net_Cons)
Fe1	netif name: eth5 [Eth src] 99:88:77:66:55:44 [Eth dest] 9:8:7:6:5:4 [Eth type] IP Length of pbuf = 42 Total length of pbuf + chain = 42 IP packets sent.... Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22 Protocol: 1 (ICMP), ICMP Type: 8 (Echo), ICMP Code: 0 Identifier: 16811 no of bytes written = 42 Ethernet IP frame sent	Error: Invalid parameter. buffer pointer is null.
Fe2	netif name: eth5 [Eth src] 99:88:77:66:55:44 [Eth dest] 9:8:7:6:5:4 [Eth type] IP Length of pbuf = 42	Error: Invalid parameter. num_read pointer is null.

	Total length of pbuf + chain = 42 IP packets sent.... Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22 Protocol: 1 (ICMP), ICMP Type: 8 (Echo), ICMP Code: 0 Identifier: 16811 no of bytes written = 42 Ethernet IP frame sent	
Fe3	netif name: eth5 [Eth src] 99:88:77:66:55:44 [Eth dest] 9:8:7:6:5:4 [Eth type] IP Length of pbuf = 42 Total length of pbuf + chain = 42 IP packets sent.... Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22 Protocol: 1 (ICMP), ICMP Type: 8 (Echo), ICMP Code: 0 Identifier: 16811 no of bytes written = 42 Ethernet IP frame sent	Error: Invalid parameter. Unknown ICMP type.
Fe4	netif name: eth5 [Eth src] 99:88:77:66:55:44 [Eth dest] 9:8:7:6:5:4 [Eth type] IP Length of pbuf = 42 Total length of pbuf + chain = 42 IP packets sent.... Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22 Protocol: 1 (ICMP), ICMP Type: 8 (Echo), ICMP Code: 0 Identifier: 16811 no of bytes written = 42 Ethernet IP frame sent	Eth [IP] frame added to list begin ip_input_verify() ip_input_verify() returns ERR_OK end ip_input_verify() LPSK_ICMPTYPE_DATA num of bytes read = 42 IP packets received.... Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22 Protocol: 1 (ICMP), ICMP Type: 8 (Echo), ICMP Code: 0 Identifier: 16811
Fe5	netif name: eth5 [Eth src] 99:88:77:66:55:44 [Eth dest] 9:8:7:6:5:4 [Eth type] IP Length of pbuf = 42 Total length of pbuf + chain = 42 IP packets sent.... Version: 4, Src IP: 192.168.0.11,	Eth [IP] frame added to list begin ip_input_verify() ip_input_verify() returns ERR_OK end ip_input_verify() IP packets received.... Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22 Protocol: 1 (ICMP), ICMP Type: 8

	<p>Dest IP: 192.168.0.22  Protocol: 1 (ICMP), ICMP Type: 8  (Echo), ICMP Code: 0  Identifier: 16811  no of bytes written = 42  Ethernet IP frame sent</p>	<p>(Echo), ICMP Code: 0  Identifier: 16811  begin icmp_input()  no of bytes written = 42  Unable to get ARP reply from destination host. IP packet dropped.  end icmp_input()</p>
Fe6	<p>netif name: eth5  [Eth src] 99:88:77:66:55:44  [Eth dest] 9:8:7:6:5:4  [Eth type] IP  Length of pbuf = 42  Total length of pbuf + chain = 42  IP packets sent....  Version: 4, Src IP: 192.168.0.11,  Dest IP: 192.168.0.22  Protocol: 1 (ICMP), ICMP Type: 0  (Echo Reply), ICMP Code: 0  Identifier: 16811  no of bytes written = 42  Ethernet IP frame sent</p>	<p>Eth [IP] frame added to list  begin ip_input_verify()  ip_input_verify() returns ERR_OK  end ip_input_verify()  Warning: ICMP Echo Reply ID mismatched.  LPSK_ICMPTYPE_DATA  num of bytes read = 42  IP packets received....  Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22  Protocol: 1 (ICMP), ICMP Type: 0  (Echo Reply), ICMP Code: 0  Identifier: 16811</p>
Fe7	<p>netif name: eth5  [Eth src] 99:88:77:66:55:44  [Eth dest] 9:8:7:6:5:4  [Eth type] IP  Length of pbuf = 42  Total length of pbuf + chain = 42  IP packets sent....  Version: 4, Src IP: 192.168.0.11,  Dest IP: 192.168.0.22  Protocol: 1 (ICMP), ICMP Type: 0  (Echo Reply), ICMP Code: 0  Identifier: 16822  no of bytes written = 42  Ethernet IP frame sent</p>	<p>Eth [IP] frame added to list  begin ip_input_verify()  ip_input_verify() returns ERR_OK  end ip_input_verify()  ICMP Echo Reply ID matched.  LPSK_ICMPTYPE_DATA  num of bytes read = 42  IP packets received....  Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22  Protocol: 1 (ICMP), ICMP Type: 0  (Echo Reply), ICMP Code: 0  Identifier: 16822</p>

Function Test Group F: <i>recv_ping()</i>		
Test ID	Expected Result (Net_Prod)	Expected Result (Net_Cons)
Ff1	netif name: eth5 [Eth src] 99:88:77:66:55:44 [Eth dest] 9:8:7:6:5:4 [Eth type] IP Length of pbuf = 42 Total length of pbuf + chain = 42 IP packets sent.... Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22 Protocol: 1 (ICMP), ICMP Type: 0 (Echo Reply), ICMP Code: 0 Identifier: 16822 no of bytes written = 42 Ethernet IP frame sent	Error: Invalid parameter. buffer pointer is null.
Ff2	netif name: eth5 [Eth src] 99:88:77:66:55:44 [Eth dest] 9:8:7:6:5:4 [Eth type] IP Length of pbuf = 42 Total length of pbuf + chain = 42 IP packets sent.... Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22 Protocol: 1 (ICMP), ICMP Type: 0 (Echo Reply), ICMP Code: 0 Identifier: 16822 no of bytes written = 42 Ethernet IP frame sent	Error: Invalid parameter. num_read is null.
Ff3	netif name: eth5 [Eth src] 99:88:77:66:55:44 [Eth dest] 9:8:7:6:5:4 [Eth type] IP Length of pbuf = 42 Total length of pbuf + chain = 42 IP packets sent.... Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22 Protocol: 1 (ICMP), ICMP Type: 0 (Echo Reply), ICMP Code: 0 Identifier: 16822 no of bytes written = 42	Eth [IP] frame added to list begin ip_input_verify() ip_input_verify() returns ERR_OK end ip_input_verify() ICMP Echo Reply ID matched. NO_ERROR num of bytes read = 28 IP packets received.... Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22 Protocol: 1 (ICMP), ICMP Type: 0 (Echo Reply), ICMP Code: 0 Identifier: 16822

	Ethernet IP frame sent	
--	------------------------	--

Function Test Group G: <i>get_eth_arp_frame()</i>		
Test ID	Expected Result (Net_Prod)	Expected Result (Net_Cons)
Fg1	<p>netif name: eth5  [Eth src] 99:88:77:66:55:44  [Eth dest] 9:8:7:6:5:4  [Eth type] IP  Length of pbuf = 42  Total length of pbuf + chain = 42  IP packets sent....  Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22  Protocol: 17 (UDP)  no of bytes written = 42  Ethernet IP frame sent</p> <p>netif name: eth5  [Eth src] 11:22:33:44:55:11  [Eth dest] 1:2:3:4:5:1  [Eth type] ARP  Length of pbuf = 14  Total length of pbuf + chain = 14  no of bytes written = 14  Ethernet ARP frame sent</p>	<p>Eth [IP] frame added to list  get_next_frame:: skip ethernet_input()  Eth [ARP] frame added to list  get_eth_arp_frame:: begin  ethernet_input()  get_eth_arp_frame:: ethernet_input()  returns ERR_OK  get_eth_arp_frame:: end  ethernet_input()  NO_ERROR</p>

Function Test Group H: <i>ip_input_verify()</i>		
Test ID	Expected Result (Net_Prod)	Expected Result (Net_Cons)
Fh1	<p>netif name: eth5  [Eth src] 99:88:77:66:55:44  [Eth dest] 9:8:7:6:5:4  [Eth type] IP  Length of pbuf = 42  Total length of pbuf + chain = 42  IP packets sent....  Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22  Protocol: 1 (ICMP), ICMP Type: 0 (Echo Reply), ICMP Code: 0  Identifier: 16822  no of bytes written = 42  Ethernet IP frame sent</p> <p>netif name: eth5  [Eth src] 11:22:33:44:55:11  [Eth dest] 1:2:3:4:5:1  [Eth type] ARP  Length of pbuf = 14  Total length of pbuf + chain = 14  no of bytes written = 14  Ethernet ARP frame sent</p>	<p>Eth [IP] frame added to list  begin ethernet_input()  ethernet_input() returns ERR_OK  end ethernet_input()  LPSK_ETHTYPE_DATA  num of bytes read = 42  netif name: eth6  IP packet received....  Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22  Protocol: 1 (ICMP), ICMP Type: 0 (Echo Reply), ICMP Code: 0  Identifier: 16822</p> <p>begin ip_input_verify()  Error: Invalid parameter. pbuf pointer is null.  end ip_input_verify()</p>
Fh2	<p>netif name: eth5  [Eth src] 99:88:77:66:55:44  [Eth dest] 9:8:7:6:5:4  [Eth type] IP  Length of pbuf = 42  Total length of pbuf + chain = 42  IP packets sent....  Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22  Protocol: 1 (ICMP), ICMP Type: 0 (Echo Reply), ICMP Code: 0  Identifier: 16822  no of bytes written = 42  Ethernet IP frame sent</p> <p>netif name: eth5  [Eth src] 11:22:33:44:55:11  [Eth dest] 1:2:3:4:5:1</p>	<p>Eth [IP] frame added to list  begin ethernet_input()  ethernet_input() returns ERR_OK  end ethernet_input()  LPSK_ETHTYPE_DATA  num of bytes read = 42  netif name: eth6  IP packet received....  Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22  Protocol: 1 (ICMP), ICMP Type: 0 (Echo Reply), ICMP Code: 0  Identifier: 16822</p> <p>begin ip_input_verify()  Error: Invalid parameter. netif pointer is null.  end ip_input_verify()</p>

	<p>[Eth type] ARP  Length of pbuf = 14  Total length of pbuf + chain = 14  no of bytes written = 14  Ethernet ARP frame sent</p>	
Fh3	<p>netif name: eth5  [Eth src] 99:88:77:66:55:44  [Eth dest] 9:8:7:6:5:4  [Eth type] IP  Length of pbuf = 42  Total length of pbuf + chain = 42  IP packets sent....  Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22  Protocol: 1 (ICMP), ICMP Type: 0 (Echo Reply), ICMP Code: 0  Identifier: 16822  no of bytes written = 42  Ethernet IP frame sent</p> <p>netif name: eth5  [Eth src] 11:22:33:44:55:11  [Eth dest] 1:2:3:4:5:1  [Eth type] ARP  Length of pbuf = 14  Total length of pbuf + chain = 14  no of bytes written = 14  Ethernet ARP frame sent</p>	<p>Eth [IP] frame added to list  begin ethernet_input()  ethernet_input() returns ERR_OK  end ethernet_input()  LPSK_ETHTYPE_DATA  num of bytes read = 42  netif name: eth6  IP packet received....  Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22  Protocol: 1 (ICMP), ICMP Type: 0 (Echo Reply), ICMP Code: 0  Identifier: 16822  Warning: IP header length does not fit in pbuf length. IP packet dropped.</p>
Fh4	<p>netif name: eth5  [Eth src] 99:88:77:66:55:44  [Eth dest] 9:8:7:6:5:4  [Eth type] IP  Length of pbuf = 42  Total length of pbuf + chain = 42  IP packets sent....  Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22  Protocol: 1 (ICMP), ICMP Type: 0 (Echo Reply), ICMP Code: 0  Identifier: 16822  no of bytes written = 42  Ethernet IP frame sent</p> <p>netif name: eth5  [Eth src] 11:22:33:44:55:11</p>	<p>Eth [IP] frame added to list  begin ethernet_input()  ethernet_input() returns ERR_OK  end ethernet_input()  LPSK_ETHTYPE_DATA  num of bytes read = 42  netif name: eth6  IP packet received....  Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22  Protocol: 1 (ICMP), ICMP Type: 0 (Echo Reply), ICMP Code: 0  Identifier: 16822  Warning: IP total length does not fit in pbuf length. IP packet dropped.</p>

	<p>[Eth dest] 1:2:3:4:5:1  [Eth type] ARP  Length of pbuf = 14  Total length of pbuf + chain = 14  no of bytes written = 14  Ethernet ARP frame sent</p>	
Fh5	<p>netif name: eth5  [Eth src] 99:88:77:66:55:44  [Eth dest] 9:8:7:6:5:4  [Eth type] IP  Length of pbuf = 42  Total length of pbuf + chain = 42  IP packets sent....  Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.99  Protocol: 1 (ICMP), ICMP Type: 0 (Echo Reply), ICMP Code: 0  Identifier: 16822  no of bytes written = 42  Ethernet IP frame sent</p> <p>netif name: eth5  [Eth src] 11:22:33:44:55:11  [Eth dest] 1:2:3:4:5:1  [Eth type] ARP  Length of pbuf = 14  Total length of pbuf + chain = 14  no of bytes written = 14  Ethernet ARP frame sent</p>	<p>Eth [IP] frame added to list  begin ethernet_input()  ethernet_input() returns ERR_OK  end ethernet_input()  LPSK_ETHTYPE_DATA  num of bytes read = 42  netif name: eth6  IP packet received....  Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.99  Protocol: 1 (ICMP), ICMP Type: 0 (Echo Reply), ICMP Code: 0  Identifier: 16822  begin ip_forward()  [ip_forward] netif-&gt;output()  no of bytes written = 42  Unable to get ARP reply from destination host. IP packet dropped.  end ip_forward()</p>
Fh6	<p>netif name: eth5  [Eth src] 99:88:77:66:55:44  [Eth dest] 9:8:7:6:5:4  [Eth type] IP  Length of pbuf = 42  Total length of pbuf + chain = 42  IP packets sent....  Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.2.111  Protocol: 1 (ICMP), ICMP Type: 0 (Echo Reply), ICMP Code: 0  Identifier: 16822  no of bytes written = 42  Ethernet IP frame sent</p> <p>netif name: eth5</p>	<p>Eth [IP] frame added to list  begin ethernet_input()  ethernet_input() returns ERR_OK  end ethernet_input()  LPSK_ETHTYPE_DATA  num of bytes read = 42  netif name: eth6  IP packet received....  Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.2.111  Protocol: 1 (ICMP), ICMP Type: 0 (Echo Reply), ICMP Code: 0  Identifier: 16822  begin ip_forward()  No forwarding route for dest IP address.  end ip_forward()</p>

	[Eth src] 11:22:33:44:55:11 [Eth dest] 1:2:3:4:5:1 [Eth type] ARP Length of pbuf = 14 Total length of pbuf + chain = 14 no of bytes written = 14 Ethernet ARP frame sent	
Fh7	netif name: eth5 [Eth src] 99:88:77:66:55:44 [Eth dest] 9:8:7:6:5:4 [Eth type] IP Length of pbuf = 42 Total length of pbuf + chain = 42 IP packets sent.... Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22 Protocol: 1 (ICMP), ICMP Type: 0 (Echo Reply), ICMP Code: 0 Identifier: 16822 no of bytes written = 42 Ethernet IP frame sent <p>netif name: eth5  [Eth src] 11:22:33:44:55:11  [Eth dest] 1:2:3:4:5:1  [Eth type] ARP  Length of pbuf = 14  Total length of pbuf + chain = 14  no of bytes written = 14  Ethernet ARP frame sent</p>	Eth [IP] frame added to list begin ethernet_input() ethernet_input() returns ERR_OK end ethernet_input() LPSK_ETHTYPE_DATA num of bytes read = 42 netif name: eth6 IP packet received....  Checksum failed. IP packet dropped.
Fh8	netif name: eth5 [Eth src] 99:88:77:66:55:44 [Eth dest] 9:8:7:6:5:4 [Eth type] IP Length of pbuf = 42 Total length of pbuf + chain = 42 IP packets sent.... Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22 Protocol: 1 (ICMP), ICMP Type: 0 (Echo Reply), ICMP Code: 0 Identifier: 16822 no of bytes written = 42 Ethernet IP frame sent	Eth [IP] frame added to list begin ethernet_input() ethernet_input() returns ERR_OK end ethernet_input() LPSK_ETHTYPE_DATA num of bytes read = 42 netif name: eth6 IP packet received....  Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22 Protocol: 1 (ICMP), ICMP Type: 0 (Echo Reply), ICMP Code: 0 Identifier: 16822 begin ip_input_verify() ip_input_verify() returns ERR_OK

netif name: eth5  
[Eth src] 11:22:33:44:55:11  
[Eth dest] 1:2:3:4:5:1  
[Eth type] ARP  
Length of pbuf = 14  
Total length of pbuf + chain = 14  
no of bytes written = 14  
Ethernet ARP frame sent

end ip\_input\_verify()

Function Test Group I: <i>etharp_request()</i>		
Test ID	Expected Result (Net_Prod)	Expected Result (Net_Cons)
Fi1	Error: Invalid parameter. netif pointer address is null.	(none)
Fi2	Error: Invalid parameter. IP address is null.	(none)
Fi3	ARP table before etharp_request() ip_addr: 0.0.0.0 ... MAC: 0:0:0:0:0:0 begin etharp_request() num of bytes written = 42 get_next_frame:: skip ethernet_input() Eth [ARP] frame added to list get_eth_arp_frame:: begin ethernet_input() get_eth_arp_frame:: ethernet_input() returns ERR_OK get_eth_arp_frame:: end ethernet_input() end etharp_request() ARP table after etharp_request() ip_addr: 192.168.0.22 ... MAC: 7:6:5:4:3:2	get_next_frame:: skip ethernet_input() Eth [ARP] frame added to list get_eth_arp_frame:: begin ethernet_input() num of bytes written = 42 get_eth_arp_frame:: ethernet_input() returns ERR_OK get_eth_arp_frame:: end ethernet_input() NO_ERROR ARP table ip_addr: 192.168.0.11 ... MAC: 6:5:4:3:2:1

Function Test Group J: <i>send_icmp()</i>		
Test ID	Expected Result (Net_Prod)	Expected Result (Net_Cons)
Fj1	Error: Invalid parameter. Unknown ICMP type or ICMP type not supported. Only support ICMP Echo message	LPSK_IPTYPE_ERROR
Fj2	Src IP: 192.168.0.11 Dest IP: 0.0.0.0 begin raw_sendto() raw_sendto: No route to destination IP addr. Packet not sent. end raw_sendto() send_icmp() returns ERR_RTE (Routing Problem)	LPSK_IPTYPE_ERROR
Fj3	Error: Invalid parameter. Unknown ICMP type or ICMP type not supported. Only support ICMP Echo message	LPSK_IPTYPE_ERROR
Fj4	Src IP: 0.0.0.0 Dest IP: 192.168.0.22 begin raw_sendto() num of bytes written = 42 get_next_frame:: skip ethernet_input() Eth [ARP] frame added to list listget_eth_arp_frame:: begin ethernet_input() num of bytes read = 42 get_eth_arp_frame:: ethernet_input() returns ERR_OK get_eth_arp_frame:: end ethernet_input() IP packet sent... Ver: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22, Protocol: 1 (ICMP), ICMP Type: 8 (Echo), Code: 0 Identifier: 16811	Eth [IP] frame added to list begin ip_input_verify() ip_input_verify() returns ERR_OK end ip_input_verify()  num of bytes read = 42 IP packets received.... Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22 Protocol: 1 (ICMP), ICMP Type: 8 (Echo), ICMP Code: 0 Identifier: 16811 begin icmp_input() IP packets sent.... Version: 4, Src IP: 192.168.0.22, Dest IP: 192.168.0.11 Protocol: 1 (ICMP), ICMP Type: 0 (Echo Reply), ICMP Code: 0 Identifier: 16811

		end icmp_input()
Fj5	Src IP: 192.168.0.11 Dest IP: 192.168.0.22 begin raw_sendto() num of bytes written = 42 get_next_frame:: skip ethernet_input() Eth [ARP] frame added to list get_eth_arp_frame:: begin ethernet_input() num of bytes read = 42 get_eth_arp_frame:: ethernet_input() returns ERR_OK get_eth_arp_frame:: end ethernet_input() IP packet sent... Ver: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22, Protocol: 1 (ICMP), ICMP Type: 8 (Echo), Code: 0 Identifier: 16811 num of bytes written = 42 end raw_sendto() send_icmp() returns ERR_OK	begin ethernet_input() ethernet_input() returns ERR_OK end ethernet_input() Eth [IP] frame added to list begin ip_input_verify() ip_input_verify() returns ERR_OK end ip_input_verify()  num of bytes read = 42 IP packets received.... Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22 Protocol: 1 (ICMP), ICMP Type: 8 (Echo), ICMP Code: 0 Identifier: 16811 begin icmp_input() IP packets sent.... Version: 4, Src IP: 192.168.0.22, Dest IP: 192.168.0.11 Protocol: 1 (ICMP), ICMP Type: 0 (Echo Reply), ICMP Code: 0 Identifier: 16811 end icmp_input()

Function Test Group K: <i>send_ping()</i>		
Test ID	Expected Result (Net_Prod)	Expected Result (Net_Cons)
Fk1	Src IP: 192.168.0.11 Dest IP: 192.168.0.22 begin raw_sendto() num of bytes written = 42 get_next_frame:: skip ethernet_input() Eth [ARP] frame added to list num of bytes read = 42 get_eth_arp_frame:: begin ethernet_input() etharp_arp_input()  get_eth_arp_frame:: ethernet_input() returns ERR_OK get_eth_arp_frame:: end ethernet_input()	begin ethernet_input() etharp_arp_input() num of bytes written = 42 ethernet_input() returns ERR_OK end ethernet_input() Eth [IP] frame added to list num of bytes read = 42 begin ip_input_verify() ip_input_verify() returns ERR_OK end ip_input_verify()  IP packets received.... Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22

	<p>IP packet sent...</p> <p>Ver: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.22,</p> <p>Protocol: 1 (ICMP), ICMP Type: 8 (Echo), Code: 0</p> <p>Identifier: 16811</p> <p>num of bytes written = 42</p> <p>end raw_sendto()</p> <p>send_icmp() returns ERR_OK</p>	<p>Protocol: 1 (ICMP), ICMP Type: 8 (Echo), ICMP Code: 0</p> <p>Identifier: 16811</p> <p>begin icmp_input()</p> <p>IP packets sent....</p> <p>Version: 4, Src IP: 192.168.0.22, Dest IP: 192.168.0.11</p> <p>Protocol: 1 (ICMP), ICMP Type: 0 (Echo Reply), ICMP Code: 0</p> <p>Identifier: 16811</p> <p>num of bytes written = 42</p> <p>end icmp_input()</p>
Fk2	<p>Src IP: 192.168.0.11</p> <p>Dest IP: 192.168.0.99</p> <p>begin raw_sendto()</p> <p>num of bytes written = 42</p> <p>Unable to get ARP reply from destination host. IP packet dropped.</p> <p>end raw_sendto()</p> <p>send_icmp() returns LPSK_ETH_ARP_ERROR</p>	<p>begin ethernet_input()</p> <p>etharp_arp_input()</p> <p>etharp_arp_input: Warning: ARP request was not for us.</p> <p>ethernet_input() returns ERR_OK</p> <p>end ethernet_input()</p>
Fk3	<p>Src IP: 192.168.0.11</p> <p>Dest IP: 192.168.0.11</p> <p>begin raw_sendto()</p> <p>**loopback**</p> <p>num of bytes written = 42</p> <p>get_next_frame:: skip ethernet_input()</p> <p>Eth [ARP] frame added to list</p> <p>get_eth_arp_frame:: begin</p> <p>ethernet_input()</p> <p>num of bytes written = 42</p> <p>get_eth_arp_frame:: ethernet_input()</p> <p>returns ERR_OK</p> <p>get_eth_arp_frame:: end</p> <p>ethernet_input()</p> <p>IP packet sent...</p> <p>Ver: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.11,</p> <p>Protocol: 1 (ICMP), ICMP Type: 8 (Echo), Code: 0</p> <p>Identifier: 16811</p> <p>num of bytes written = 42</p> <p>end raw_sendto()</p>	(none)

	<pre> send_icmp() returns ERR_OK  begin ethernet_input() ethernet_input() returns ERR_OK end ethernet_input() Eth [IP] frame added to list begin ip_input_verify() ip_input_verify() returns ERR_OK end ip_input_verify() IP packets received.... Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.11 Protocol: 1 (ICMP), ICMP Type: 8 (Echo), ICMP Code: 0 Identifier: 16811 begin icmp_input() **loopback** IP packets sent.... Version: 4, Src IP: 192.168.0.11, Dest IP: 192.168.0.11 Protocol: 1 (ICMP), ICMP Type: 0 (Echo Reply), ICMP Code: 0 Identifier: 16811 num of bytes written = 42 end icmp_input() </pre>	
Fk4	Src IP: 192.168.0.11 Dest IP: 192.168.2.111 begin raw_sendto() raw_sendto: No route to dest IP address. Packet not sent. end raw_sendto() send_icmp() returns ERR_RTE (Routing Problem)	LPSK_IPTYPE_ERROR

### C. Acceptance Test Expected Results

This section lists the expected results of Net\_Prod and Net\_Cons partitions for each acceptance test.

Acceptance Test A1	
Expected Result (Net_Prod)	Expected Result (Net_Cons)
[192.168.0.11] Ping 192.168.0.22 ... [192.168.0.11] IP packet sent ... Version: 4 Source IP: 192.168.0.11 Destination IP: 192.168.0.22 Protocol: 1 (ICMP) ICMP Type: 8 (Echo) Code: 0 Identifier: 16811 [192.168.0.11] IP packet received ... Version: 4 Source IP: 192.168.0.22 Destination IP: 192.168.0.11 Protocol: 1 (ICMP) ICMP Type: 0 (Echo Reply) Code: 0 Identifier: 16811	[192.168.0.22] Waiting for incoming ... [192.168.0.22] IP packet received ... Version: 4 Source IP: 192.168.0.11 Destination IP: 192.168.0.22 Protocol: 1 (ICMP) ICMP Type: 8 (Echo) Code: 0 Identifier: 16811 [192.168.0.22] IP packet sent ... Version: 4 Source IP: 192.168.0.22 Destination IP: 192.168.0.11 Protocol: 1 (ICMP) ICMP Type: 0 (Echo Reply) Code: 0 Identifier: 16811

Acceptance Test A2	
Expected Result (Net_Prod)	Expected Result (Net_Cons)
<p>[192.168.0.11] Ping 192.168.0.22 ...</p> <p>[192.168.0.11] IP packet sent ...</p> <p>Version: 4</p> <p>Source IP: 192.168.0.11</p> <p>Destination IP: 192.168.0.22</p> <p>Protocol: 1 (ICMP)</p> <p>ICMP Type: 8 (Echo)</p> <p>Code: 0</p> <p>Identifier: 16811</p> <p>[192.168.0.11] IP packet received ...</p> <p>Version: 4</p> <p>Source IP: 192.168.0.22</p> <p>Destination IP: 192.168.0.11</p> <p>Protocol: 1 (ICMP)</p> <p>ICMP Type: 8 (Echo)</p> <p>Code: 0</p> <p>Identifier: 16822</p> <p>[192.168.0.11] IP packet sent ...</p> <p>Version: 4</p> <p>Source IP: 192.168.0.11</p> <p>Destination IP: 192.168.0.22</p> <p>Protocol: 1 (ICMP)</p> <p>ICMP Type: 0 (Echo Reply)</p> <p>Code: 0</p> <p>Identifier: 16822</p> <p>[192.168.0.11] IP packet received ...</p> <p>Version: 4</p> <p>Source IP: 192.168.0.22</p> <p>Destination IP: 192.168.0.11</p> <p>Protocol: 1 (ICMP)</p> <p>ICMP Type: 0 (Echo Reply)</p> <p>Code: 0</p> <p>Identifier: 16811</p>	<p>[192.168.0.22] Ping 192.168.0.11 ...</p> <p>[192.168.0.22] IP packet sent ...</p> <p>Version: 4</p> <p>Source IP: 192.168.0.22</p> <p>Destination IP: 192.168.0.11</p> <p>Protocol: 1 (ICMP)</p> <p>ICMP Type: 8 (Echo)</p> <p>Code: 0</p> <p>Identifier: 16822</p> <p>[192.168.0.22] IP packet received ...</p> <p>Version: 4</p> <p>Source IP: 192.168.0.11</p> <p>Destination IP: 192.168.0.22</p> <p>Protocol: 1 (ICMP)</p> <p>ICMP Type: 8 (Echo)</p> <p>Code: 0</p> <p>Identifier: 16811</p> <p>[192.168.0.22] IP packet sent ...</p> <p>Version: 4</p> <p>Source IP: 192.168.0.22</p> <p>Destination IP: 192.168.0.11</p> <p>Protocol: 1 (ICMP)</p> <p>ICMP Type: 0 (Echo Reply)</p> <p>Code: 0</p> <p>Identifier: 16811</p> <p>[192.168.0.22] IP packet received ...</p> <p>Version: 4</p> <p>Source IP: 192.168.0.11</p> <p>Destination IP: 192.168.0.22</p> <p>Protocol: 1 (ICMP)</p> <p>ICMP Type: 0 (Echo Reply)</p> <p>Code: 0</p> <p>Identifier: 16822</p>

Acceptance Test A3	
Expected Result (Net_Prod)	Expected Result (Net_Cons)
[192.168.0.11] Ping 192.168.0.22 ... Unable to get ARP reply from destination host. IP packet dropped. send_icmp() returns <u>LSPK_ETH_ARP_ERROR</u>	(none)

## LIST OF REFERENCES

- [1] C.E. Irvine, T.E. Levin, P.C. Clark, and T.D. Nguyen, “A security architecture for transient trust,” in *Computer Security Architecture Workshop*, Fairfax, VA, October 2008.
- [2] C.E. Irvine, T.E. Levin, T.D. Nguyen, and G.W. Dinolt, “The trusted computing exemplar project,” in *Proc. 2004 IEEE Systems, Man and Cybernetics Information Assurance Workshop*, West Point, NY, June 2004, pp. 109–115.
- [3] W. R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*. New York: Addison-Wesley, 1994.
- [4] R. Braden, Requirements of Internet Hosts – Communication Layers [Online]. Available: <http://tools.ietf.org/html/rfc1122>. [Accessed October 19, 2010].
- [5] Internet Protocol Suite [Online]. Available: [http://en.wikipedia.org/wiki/Internet\\_Protocol\\_Suite](http://en.wikipedia.org/wiki/Internet_Protocol_Suite). [Accessed October 19, 2010].
- [6] A. S. Tanenbaum, *Computer Networks*. Amsterdam, The Netherlands: Prentice Hall, 2002.
- [7] IEEE 802.3 [Online]. Available: <http://www.ieee802.org/3/>. [Accessed October 19, 2010].
- [8] D.L. Shinder, *Computer Networking Essentials*. Indianapolis: Cisco Press, 2002, ch. 4, pp. 109-131.
- [9] W. R. Stevens, B. Fenner, A.M. Rudoff, *UNIX Network Programming: The sockets networking API, Volume 1* (3<sup>rd</sup> ed.). New York: Addison-Wesley, 2004.
- [10] J. Postel, Internet Control Message Protocol [Online]. Available: <http://tools.ietf.org/html/rfc792>. [Accessed October 19, 2010].
- [11] TCP/IP and tcpdump Pocket reference guide [Online]. Available: <http://www.sans.org/security-resources/tcpip.pdf>. [Accessed October 19, 2010].
- [12] The Internet Engineering Task Force (IETF) [Online]. Available: <http://www.ietf.org> [Accessed October 19, 2010].
- [13] J. P. Anderson, *Computer Security Technology Planning Study*, ESD-TR-73-51, vol. I, ESD/AFSC, Hanscom AFB, Bedford, MA., October 1972 (NTIS AD-758 206).

- [14] S. H. Ames, M. Gasser, and R. R. Schell, “Security kernel design and implementation: An introduction.” *IEEE Computer*, vol. 16, no. 7, pp. 14–22, 1983.
- [15] J. Rushby, “The design and verification of secure systems,” in *8th ACM Symposium on Operating System Principles*, 1981, vol. 15, no. 5, pp. 12–21.
- [16] T. E. Levin, C. E. Irvine, and T. D. Nguyen, “Least privilege in separation kernels,” in *E-business and Telecommunication Networks* (J. Filipe and M. S. Obaidat, eds.), vol. 9 of *Communications in Computer and Information Science*, pp. 146 – 158, Berlin: Springer, 2008.
- [17] J.H. Saltzer and M.D. Schroeder, “The Protection of Information in Operating Systems,” in *Proc. IEEE*, 1975, vol. 63(9), pp. 1278–1308.
- [18] Information Assurance Directorate, “U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness,” [Online]. Version 1.03, June 2007. Available: [http://www.commoncriteriaportal.org/files/ppfiles/pp\\_skpp\\_hr\\_v1.03.pdf](http://www.commoncriteriaportal.org/files/ppfiles/pp_skpp_hr_v1.03.pdf). [Accessed October 19, 2010].
- [19] C. M. Kozierok, *The TCP/IP guide: a comprehensive, illustrated Internet protocols reference*, San Francisco, California: No Starch Press, 2005.
- [20] P.C. Clark, D.J. Shifflett, C.E. Irvine, T.D. Nguyen, and T.E. Levin, “Trusted Computing Exemplar Least Privilege Product Functional Specification,” Naval Postgraduate School Center for Information Systems Security Studies and Research, 2010.
- [21] C.E. Irvine, T. E. Levin, P.C. Clark, and T.D. Nguyen, “A Security Architecture for Transient Trust,” in *Computer Security Architecture Workshop*, October 2008, Fairfax, VA.
- [22] T.E. Levin, C.E. Irvine, T.V. Benzel, G. Bhaskara, P.C. Clark, and T.D. Nguyen, “Design Principles and Guidelines for Security.” NPS-CS-07-014, Naval Postgraduate School, Monterey, CA, November 2007.
- [23] uIP – A TCP/IP protocol stack for small 8-bit and 16-bit microcontrollers [Online]. Available: <http://www.dunkels.com/adam/uip/>. [Accessed October 22, 2010].
- [24] lwIP – A Lightweight TCP/IP Stack [Online]. Available: <http://savannah.nongnu.org/projects/lwip/>. [Accessed October 15, 2010].
- [25] TinyTCP [Online]. Available: <http://www.unusualresearch.com/tinytcp/tinytcp.htm>. [Accessed October 22, 2010].

- [26] uC/IP – The uC/IP Project [Online]. Available: <http://ucip.sourceforge.net/> [Accessed October 22, 2010].
- [27] A. Dunkels, Design and Implementation of the LWIP TCP/IP Stack [Online]. Available: <http://www.sics.se/~adam/lwip/doc/lwip.pdf>. [Accessed October 21, 2010].
- [28] D. D. Clark. Modularity and efficiency in protocol implementation [Online]. Available: <http://tools.ietf.org/html/rfc817>. [Accessed October 15, 2010].
- [29] Netif Struct Reference [Online]. Available: <http://www.nongnu.org/lwip/structnetif.html>. [Accessed October 21, 2010].
- [30] LWIP Source Download [Online]. Available: <http://download.savannah.gnu.org/releases/lwip/>. [Accessed October 15, 2010].
- [31] VMware Workstation [Online]. Available: <http://www.vmware.com/products/workstation/>. [Accessed October 15, 2010].
- [32] Windows 7 Professional [Online]. Available: <http://www.microsoft.com/windows/windows-7>. [Accessed October 15, 2010].
- [33] Fedora Core Project [Online]. Available: <http://fedoraproject.org/>. [Accessed October 15, 2010].
- [34] Open Watcom Main Page [Online]. Available: <http://openwatcom.org>. [Accessed October 15, 2010].
- [35] Ng Yeow Cheng, “Applications to support Normal and Critical Operations in a tactical MLS System,” Master’s thesis. Naval Postgraduate School, Monterey California, December 2010.
- [36] N.Doraswamy and D. Harkins, *IPSec: The New Security Standard for the Internet, Intranets, and Virtual Private Network* (2<sup>nd</sup> ed.). Upper Saddle River, NJ: Prentice Hall, 2003, pp. 43–58.

**THIS PAGE INTENTIONALLY LEFT BLANK**

## **INITIAL DISTRIBUTION LIST**

1. Defense Technical Information Center  
Ft. Belvoir, VA
2. Dudley Knox Library  
Naval Postgraduate School  
Monterey, CA
3. Kris Britton  
National Security Agency  
Fort Meade, MD
4. John Campbell  
National Security Agency  
Fort Meade, MD
5. Deborah Cooper  
DC Associates, LLC  
Reston, VA
6. Grace Crowder  
NSA  
Fort Meade, MD
7. Louise Davidson  
National Geospatial Agency  
Bethesda, MD
8. Vincent J. DiMaria  
National Security Agency  
Fort Meade, MD
9. Rob Dobry  
NSA  
Fort Meade, MD
10. Jennifer Guild  
SPAWAR  
Charleston, SC
11. CDR Scott Heller  
SPAWAR  
Charleston, SC

12. Dr. Steven King  
ODUSD  
Washington, DC
13. Steve LaFountain  
NSA  
Fort Meade, MD
14. Dr. Greg Larson  
IDA  
Alexandria, VA
15. Dr. Carl Landwehr  
National Science Foundation  
Arlington, VA
16. Dr. John Monastra  
Aerospace Corporation  
Chantilly, VA
17. John Mildner  
SPAWAR  
Charleston, SC
18. Dr. Victor Piotrowski  
National Science Foundation  
Arlington, VA
19. Jim Roberts  
Central Intelligence Agency  
Reston, VA
20. Ed Schneider  
IDA  
Alexandria, VA
21. Mark Schneider  
NSA  
Fort Meade, MD
22. Keith Schwalm  
Good Harbor Consulting, LLC  
Washington, DC

23. Ken Shotting  
NSA  
Fort Meade, MD
24. Dr. Ralph Wachter  
ONR  
Arlington, VA
25. Dr. Cynthia E. Irvine  
Naval Postgraduate School  
Monterey, CA
26. David Shifflett  
Naval Postgraduate School  
Monterey, CA
27. Professor Yeo Tat Soon  
Temasek Defence Systems Institute (TDSI)  
National University of Singapore  
Singapore
28. Tan Lai Poh  
Temasek Defence Systems Institute (TDSI)  
National University of Singapore  
Singapore
29. Ho Liang Yoong  
Singapore Technologies Engineering  
Singapore